

Holistic debugging — enabling instruction set simulation for software quality assurance

Lars Albertsson
Swedish Institute of Computer Science
SE-164 29 Kista
Sweden
lalle@sics.se

Abstract

We present *holistic debugging*, a novel method for observing execution of complex and distributed software. It builds on an instruction set simulator, which provides reproducible experiments and non-intrusive probing of state in a distributed system. Instruction set simulators, however, only provide low-level information, so a holistic debugger contains a translation framework that maps this information to higher abstraction level observation tools, such as source code debuggers.

We have created *Nornir*, a proof-of-concept holistic debugger, built on the simulator *Simics*. For each observed process in the simulated system, *Nornir* creates an abstraction translation stack, with virtual machine translators that map machine-level storage contents (e.g. physical memory, registers) provided by *Simics*, to application-level data (e.g. virtual memory contents) by parsing the data structures of operating systems and virtual machines. *Nornir* includes a modified version of the GNU debugger (*GDB*), which supports non-intrusive symbolic debugging of distributed applications. *Nornir*'s main interface is a debugger shepherd, a programmable interface that controls multiple debuggers, and allows users to coherently inspect the entire state of heterogeneous, distributed applications. It provides a robust observation platform for construction of new observation tools.

1. Introduction

Every year, the size and complexity of computer software systems increase — we build larger applications by stacking more software construction tools, such as compilers, components, runtime systems, middleware, code generators. Unfortunately, software quality assurance methods do not scale in the same manner. As the size of soft-

ware projects grow, testing and debugging takes more effort in comparison to programming, and for complex software projects, quality assurance is the dominant development cost [17]. This ratio is likely to increase further in the near future, as the proliferation of processors with multiple cores will force many software vendors to write parallel programs, which are harder to test, profile, and debug than sequential programs. Hence, there is a desperate need for scalable quality assurance methods that handle concurrency errors.

Test case execution combined with program observation, e.g. debugging, white-box testing, tracing, and performance profiling, is the predominant software quality assurance method today. Unlike software construction tools, the observation tools available are generally not stackable nor capable of coherent information exchange with other observation tools. Most existing software observation tools implement their own probing mechanisms, and are limited to observing a single abstraction layer in a homogeneous environment. Therefore, they only partially address their observation needs for complex software, and the oldest and most primitive debugging method — the print statement and variations thereof — is still predominant.

We claim that the the inherently fragile observation technologies that existing test and debug tools rely on is an explanation for their lack of scalability. The factors contributing to the fragility have been discussed in several papers [10, 16, 21, 24] as inherent difficulties in debugging distributed software: *indeterminism* (also referred to as non-repeatability, nonreproducibility), *probe effect* (aka intrusion), and *incomplete causal ordering*. If an observation technique cannot provide repeatable experiments, and if the act of probing affects the observation, it is hard to use it for building scalable observation tools. The factors are discussed in more detail in Section 2.1. Although these factors are particularly troublesome when observing distributed software, they occur in any software using asyn-

chronous services, e.g. clocks, interrupts, Unix signals, non-blocking I/O, etc, and limit observability in any complex computer system.

1.1. Simulation

The problems mentioned above are not unique to software engineering, but occur in all natural science engineering disciplines; all physical systems are indeterministic to some extent and affected by probe effect. Engineers in other fields often use simulation, a technique unaffected by these problems, as a complement to real experiments. It has not yet become widespread for software engineering, however.

Using simulators for observing software systems has major benefits: distributed systems can be observed without intrusion, and experiments can be repeated. Moreover, a single-threaded simulator implementation executes the processes in a distributed system in a deterministic order, and events are therefore globally ordered in a single serialisation of concurrent events.

Instruction set simulators could be useful for observing software systems, but they have one property that prevents them from being useful for most software: they are only able to provide information at their abstraction level: the hardware/software boundary. They can be probed for software-visible hardware state, i.e. physical memory contents, register contents, etc, but have little or no knowledge about the programs running in the simulated machine, and are not particularly useful for probing high-level constructs, such as variable contents in user-space programs or database tables.

1.2. Holistic debugging

Our main contribution is holistic debugging, a novel method for observing complex computer software running in instruction set simulators. A holistic debugger provides a translation framework that maps low-level data probed from the simulator to source-level application data. It also includes symbolic debuggers for inspecting individual processes in a simulated system. The debuggers are controlled by a *debugger shepherd*, which supports coherent observation of all participating processes in a distributed system. The shepherd is programmable and allows users to create new observation tools and debugging abstractions, and to write application-specific surveillance routines.

A holistic debugger should not be thought of as yet another tool that solves a particular, narrow problem better than other tools. Although it can be used as an interactive debugger, its primary purpose is to serve as a meta-tool that enables construction of new tools, based on more robust techniques than existing tools.

We have demonstrated that building a holistic debugger is feasible by creating the prototype Nornir, built on the

complete system simulator Simics [20]. Building a holistic debugger of reasonable quality, with support for many flavours of architectures, operating systems, and programming languages, is a huge task, beyond the scope of a research project, and Nornir implements a subset of a holistic debugger.

2. Background

2.1. Issues with observing distributed systems

Indeterminism Modern computers are indeterministic. There are factors affecting program execution that cannot be accurately predicted, for example interrupt arrival times, memory communication interleavings, subroutine execution times, and clock readings. Complex programs are always affected by such random factors, and program executions are therefore not fully reproducible, unless the program is explicitly designed to be independent of unpredictable factors. In theory, repeatable execution is a prerequisite for the standard repetitive debugging procedure. In practice, repetitive debugging is meaningful for simple programs, as long as the variations are small. Indeterministic execution, however, effectively prevents construction of scalable observation tools. Development and use of automated tools when experiments cannot be reliably reproduced is usually too time-consuming to be worthwhile.

Probe effect Any attempt to monitor a computer system with software probes will change the system's behaviour. This is referred to as *probe effect* [10]. The probe effect contributes to the indeterminism problem, and also limits the amount of data that can be observed in a running system. A monitoring service that suffers from probe effect provides a service whose quality degrades with increased usage. Such a service is inherently fragile and unsuitable for scalable observation.

Incomplete causal ordering There is no global clock in distributed systems, and a global ordering on all events in a system can often not be determined, even in post-mortem. Observing a partial ordering defined by the happens-before relation [18], however, is sufficient for observing the execution of a distributed system. In order for a tool to observe this partial ordering, it must be able to observe all messages sent between processes, and their points of arrival. This can be difficult in practice. In some distributed systems, messages and arrival points are straightforward to record, as in the case of network packets delivered to an application. Other types of messages, such as hardware cache transfers, are difficult to observe, and building tools that record and replay distributed executions involving such messages is hard.

We have not found an established name for this problem, and will call it *incomplete causal ordering*.

2.2. Complete system simulation

The issues mentioned above can be avoided by using a simulator, and a holistic debugger can be built on any deterministic simulator that provides good programming and observation services. Instruction set simulators have suitable characteristics to serve as debugging platforms: They provide models that are detailed enough for running applications in binary form, but still run sufficiently fast to run large applications. We have built our prototype implementation on Simics [20], a type of instruction set simulator known as complete system simulator [23] (or sometimes full system simulator).

Simics is binary compatible with commodity computers, and includes models of processor, memory, disks, network cards, etc. It runs unmodified commodity operating systems and applications in binary form. Simics can simulate multiprocessor machines and multiple networked computers, which may be heterogeneous in architecture. Simics is designed to be deterministic; if a simulation scenario starts in a well defined simulation state, and the input model is synthetic and predictable, it will produce exactly the same scenario for each simulation.

Simics runs roughly two orders of magnitude slower than the host machine when modelling a machine similar to the host. It is slow enough to be a significant drawback for simulation as a method, but fast enough to allow large applications to be observed. The simulation speed depends on the accuracy of the timing model. We usually run with a coarse model, where every instruction takes one clock cycle. If a more accurate timing model is desired, Simics allows users to model cache memory hierarchies, providing a good timing approximation without sacrificing much simulation performance. It also supports detailed models of processor pipelines, out-of-order execution, and speculative execution, at the price of severe performance degradation. Magnus Ekman's dissertation [9] contains some benchmarks on Simics with different timing models. Complete system simulator timing models have been validated and discussed by Gibson et.al. [11].

3. Observing simulated software

Holistic debugging takes a complete system perspective on distributed system observation. A holistic debugger runs a distributed software system in a simulator and provides the user with means to examine all components in a system simultaneously, at any abstraction level higher than the simulator's level.

A complete system simulator provides non-intrusive access to all system state visible to software. Unlike standard debuggers, which use probing services supplied by the runtime system to probe the state of running processes, the holistic debugger must use non-intrusive probing techniques, and cannot rely on runtime system services. It probes the simulator for machine state, but the information retrieved is raw, binary information that has been transformed by compilers, virtual machines, and operating systems, and is no longer easily comprehensible to humans. In order to make this information useful for a programmer, it must be translated back to the abstraction level the programmer deals with, i.e. to variables and types in the programming languages used in the application.

3.1. Abstraction stacks

Each program in a computer system runs in a machine, which interprets the program instructions and updates machine state accordingly. The most basic machine is the physical machine, where instructions are interpreted by hardware, and machine state is stored in physical storage, such as memory, disk, and registers. Each machine has a set of instructions that programs can use, and programmers use a compiler to translate source code into the machine's instruction set. A physical computer usually runs only one program directly on the hardware, and in many cases, this program is an operating system. The operating system provides virtual machines, in which other programs can run. The programs in the virtual machines are likewise programmed in a high-level language, translated by a compiler to machine instructions. Some of these programs may in turn form other types of virtual machines, interpreting some program, which may be generated by a compiler, and so on. Computer systems generally contain a number of such abstraction stacks, seldom more than a few levels deep.

For each program in a stack, there is a symbolic transformation, where a compiler transforms source code to machine code. This is in theory a one-way transformation, but most compilers provide debugging information that contains adequate information to perform reverse translation, even in the presence of compiler optimisations.

For each virtual machine in the stack, there is also a machine transformation; the storage of the program running in the virtual machine is mapped to storage in the machine that is running the program providing the virtual machine. For example, the virtual memory and registers of the virtual machine corresponding to a Unix process is mapped to physical registers, memory, or disk blocks. The machine transformation is usually reversible, if the state of the virtual machine can be examined.

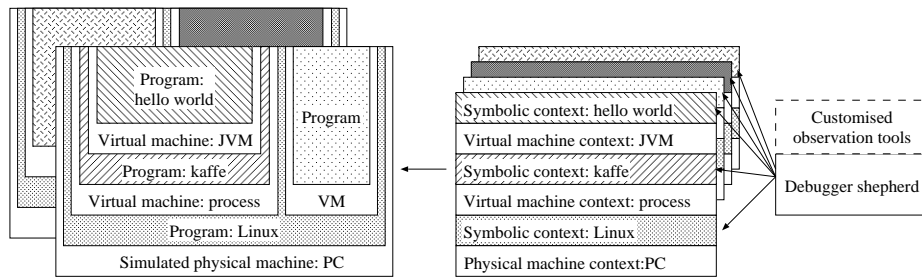


Figure 1. Holistic debugger structure with example applications.

3.2. Translation stacks

In a holistic debugger, for each inspected process in the simulated system, there is an associated *abstraction translation stack*. A translation stack consists of pairs of symbolic context probes and machine context probes, corresponding to the symbolic transformations and machines of the inspected process. The structure is shown in Figure 1. When the user inspects a particular program, a translation stack is instantiated. It includes a symbolic probe that lets the user inspect the execution and state of the program, similarly to a standard debugger. The symbolic probe queries the underlying machine context probe for program state data.

Machine context probes that refer to a physical machine query the simulator for simulated machine state. Machine context probes that refer to virtual machines, for example operating system processes, include a virtual machine translator (VMT) — a component that translates requests for virtual machine state to state requests to the underlying machine context probe. In order for the VMT to perform storage reference translations, it queries the state of the program providing the virtual machine, using its symbolic context probe.

There are no fundamental problems stacking probes in this manner, as long as the necessary information for performing reverse translations is available. The stacked translator design enables translation of the information available in the underlying simulators to any higher abstraction level in the system.

3.3. Abstraction translation in Nornir

Nornir currently supports observation of programs in simulated Linux systems. We have modified the GNU debugger (GDB) to support observation of simulated computers; a specialised target backend in GDB implements a number of debugging primitives that are necessary for GDB to operate. The Nornir GDB target backend connects and sends requests corresponding to basic debugging primitives (reading memory, inserting breakpoints, etc) over a TCP/IP socket to a *debugger broker*, a library loaded into Simics.

The debugger broker acts administrative hub for the GDB processes: It instantiates a machine context probe, including a virtual machine translator (VMT), for each process debugged, and connects it to a GDB socket. It controls simulation progress, and only allows simulation to proceed when all GDB instances have issued *continue* commands. Nornir’s main components are shown in Figure 2.

A Unix process machine context probe implements the debugging primitives required by GDB. Registers and memory addresses in GDB refer to virtual registers and addresses in a Linux process. The VMT has knowledge about operating system abstractions, and translates from virtual registers and addresses to physical registers and addresses by parsing the process list and virtual memory page tables in the running Linux kernel. Since GDB breakpoints refer to virtual addresses, the VMT translates and inserts breakpoints at the appropriate physical address. The virtual memory mappings may change for a breakpoint, and the VMT watches the associated page table entries for any changes and adjust the breakpoints accordingly. The VMT uses a static symbolic translator for probing the kernel state, described below.

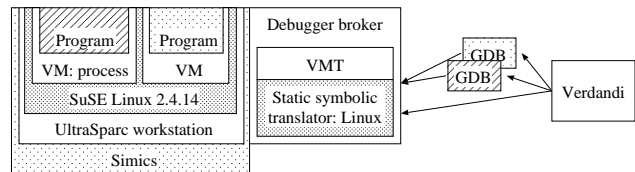


Figure 2. Nornir logical structure

3.4. Static symbolic translation

Adapting GDB to operate on a simulated target machine is a natural method for creating a symbolic probe. GDB, however, cannot easily be integrated into another program. Building a VMT on top of GDB would therefore involve retrieving information via text interface parsing, which is error-prone to program and could cause performance problems. The Linux VMT makes heavy use of the kernel sym-

bolic probe, so we want to access it with an efficient, statically typed interface. We have therefore created a static symbolic translator, which automatically generates code that matches the types in the Linux kernel.

The Normir build system compiles a Linux kernel, telling the compiler to generate debug information. The kernel is loaded into GDB, which uses the debug information to generate a report on the memory layout of the data types in Linux. The report is fed to a *target replicator*, which parses it and generates a C++ class for each type in the kernel. The replicated type classes contain methods that can read a data object of its corresponding type from a target memory address. In the case of compound data types, there are also methods for accessing the individual data members. By programming the VMT with target replication, it becomes resistant to changes in the kernel source, and only has to be changed when there is a major reorganisation of data structures, or when there are semantic changes. Over the time we have developed Normir, we have used Linux kernels ranging from the 2.1 to 2.4 series, and there have been very few such changes, requiring only small adjustments to the VMT code.

Static symbolic translation assumes that the probed data structures reside in memory, and that their address can easily be retrieved. It works sufficiently well for Linux, which uses global variables to store important data structures, but it does not work as well for probing local variables in a routine. We will eventually need to switch to a hybrid approach, where the generated type classes also have the ability to query the running kernel via GDB, thus supporting local variable probing without sacrificing type safety.

4. Debugger shepherd

The functionality described above is sufficient for observing arbitrary distributed software in a robust manner; we can instantiate a symbolic debugger for the processes in the application and debug them individually. Debugging distributed applications in this manner is an improvement over existing debug methods, since the user can examine intermittent behaviour. Having one manually controlled debugger for each process is inconvenient for debugging a large number of processes, however, and if a user wants to compare data in different processes, he must do so manually.

The main interface of a holistic debugger is therefore a *debugger shepherd* — a master inspection component responsible for instantiating and controlling translation stacks. The shepherd has access to all symbolic translators, and allows the user to follow causal paths (sequences of causally related events) by inspecting and comparing state in multiple processes.

Verdandi, Normir’s implementation of a debugger shep-

herd, is a holistic debugger interface implemented in Python. It is convenient to use a standard scripting language for the main interface, since existing code and standard libraries can be reused. Python also includes an interactive interpreter, which can be used by users that prefer interactive debugging.

Verdandi spawns a GDB process for each debugged process in the target system and uses the GDB machine interface (GDB/MI), a structured text format, for communicating with GDB. The most basic GDB functionality, such as inserting breakpoints and reading variables, is exported to Verdandi. In a typical usage scenario, the user inserts a number of eventpoints, calls a *waitFor* routine that runs the simulation until either of these occur, inspects variables, calls *waitFor* again, etc.

5. Holistic debugging abstractions

Holistic debugging adds new dimensions to debugging: time, multiple processes, and multiple abstraction levels. This opens an opportunity and need for new debugging abstractions that complement the old abstractions and match users’ needs better. We suggest a few such abstractions below. These abstractions are easy to implement if the holistic debugger architecture is sound, and users can add their own domain-specific abstractions.

5.1. Causal path monitors

Distributed application often deal with causal paths — distributed sequences of events that are causally related. Debugging distributed software often involves following causal paths with erroneous behaviour. Most programming languages and associated debuggers provide sequential, non-distributed programming models and force programmers to write and debug distributed software in terms of individual processes. This division is orthogonal to the application logic, and adds complexity to the debugging process.

The debugger shepherd provides effective means for debugging causal paths. A user that wants to debug the behaviour of a causal path can write routines that monitor the path, inspect the state of participating processes when interesting eventpoints are triggered, and alert the user of unexpected inconsistencies in the inspected data. We will refer to such a routine as a *causal path monitor*. Since a holistic debugger allows debugging of multiple abstraction layers, a causal path monitor can follow the path vertically through abstraction layers, e.g. into the operating system, as well as horizontally, across multiple processes and machines.

Verdandi implements threaded causal path monitors, where each monitor is executed as a sequential routine in a separate Python thread, and runs independently of other

monitors. The monitors call blocking routines when they want to wait for new eventpoints, and an eventpoint manager thread in the debugger shepherd controls the simulation, allowing it to proceed only when all monitors are waiting for new eventpoints. This abstraction allows causal path behaviour to be expressed as a compact sequence of statements. Threaded causal path monitors can run independently of each other, and are reusable debugging components, useful for large systems with multiple developers.

We believe that that the use of causal path monitors is more cost-efficient than interactive debugging. Writing a causal path monitor involves slightly more typing than manual debugging a path, but the monitor can be added to a project, reused by all developers in the project, and activated if the path in question is suspected of behaving erroneously again.

5.2. Eventpoints

Eventpoints is a generalisation of breakpoints — a monitor for any type of event in the target system that is of interest to the user. Time breakpoints, which trigger at a specific point in time in the simulated system, is a simple example. Eventpoints can be used to monitor lower level events than the application normally deals with, for example the reception of a network packet with a particular content, or program-specific events that are more complex than the execution of a single code statement. New types of eventpoints can be created hierarchically by recursive use of the Observer design pattern. A typical user-defined eventpoint class would run causal path monitors that inspect the paths corresponding to a high-level event, and notify the Verdandi run-time system when the event has triggered. For example, an eventpoint subclass monitoring the arrival of a particular http query can use ordinary breakpoints to monitor `read` system calls, buffer the data read on each socket, compare the concatenated buffers, and trigger when the sought string arrives.

6. The holistic debugger as a platform

Holistic debugging addresses a major debugging problem that currently has no good solution. Nevertheless, we believe that its most important potential is the robust platform it provides for building other software observation and analysis tools, much like an operating system is a solid platform for other programs. The construction of such tools lies beyond the scope of this paper, but a holistic debugger's utility as a tool platform influences our research, and the design of Nornir.

6.1. White-box testing

The properties of a holistic debugger makes it very useful for writing white-box tests, and we expect it to be an important use case. It would be particularly useful for testing applications where the coherency of the distributed state is important, e.g. peer-to-peer applications or distributed file systems, since a holistic debugger allows the user to inspect the consistency of a global application snapshot. A simulator is also useful for testing applications that are expected to survive hardware faults. In this case, a holistic debugger is not only useful for debugging errors, but also for directing fault injection to test application robustness in particularly sensitive stages, for example during online software upgrades. Today, these applications are often tested with manual fault injection, which is inefficient and expensive.

6.2. Performance analysis

A complete system simulator is a powerful tool for understanding performance behaviour of complex software [2, 13]. Since all software state is visible in a holistic debugger, it adds the capability to measure performance-related events at multiple abstraction layers, e.g. page faults, database response times, or application-specific events, and provide more detailed hints on application performance problems than standard profilers can. It is also suitable for performance analysis of soft real-time and distributed applications, since it provides non-intrusive measurements. We have used an earlier version of Nornir to demonstrate how complete system simulation, combined with virtual machine translation, can be used for performance debugging of soft real-time applications [3]. Moreover, the simulator provides global ordering, making it possible to identify time-consuming events on causal paths with undesired latency. Obtaining such information is difficult in real experiments, which is illustrated by attempts to build tools that analyse causal path performance [1, 4, 6].

7. Discussion

Using a complete system simulator as a building block for debugging tools has drawbacks, but also some major advantages; besides getting reproducible experiments, we also have good control over the execution environment. We can test and debug our software in scenarios that are difficult to arrange in practice, for example on very expensive computers, faster computers than are available today, on large numbers of low-end computers, or in the presence of hardware and communication faults.

A simulated computer is an approximation of a physical computer, and an experiment in a simulated computer is not identical to an experiment on a real computer. This

is not a significant difference from experiments on physical computers — since computers are indeterministic, no experiment is identical to another experiment.

From a software quality assurance perspective, it is not important per se whether a simulator provides an accurate model or not. It is important, however, that the conclusions we draw from experiments on simulated computers apply also for execution on physical computers, and that the flaws we hope to find and eliminate can be produced and observed in simulated systems. Whether simulation is a time- and cost-efficient quality assurance method depends on the application and on the type of errors we expect to find.

For purely logical errors that are not timing-dependent, there is little benefit of using simulation, but for timing-dependent logical errors, *race conditions*, the benefit of repeatable executions is obvious. A simulated execution does not reproduce an execution in a real system; the interleavings of events will be different for a specific experiment in a simulated system compared with a specific experiment in a real system, just as interleavings are different for two experiments in real systems. In a simulated system, however, a particular experiment can be repeated. Since the simulated timing model is inexact, it is likely that the execution will take different paths than most executions in real systems, resulting in different test coverage. Testing, however, is an ineffective method for finding concurrency errors; the event interleaving coverage that can be achieved during lab testing is small and many race conditions remain in production software. We believe that simulation will be a key technology for addressing the difficult problem of quality assurance for concurrent applications. A simulator user can control the timing model, and inject timing chaos in order to provoke intermittent errors during testing. This would enable meaningful testing for concurrency errors, and we believe that holistic debugging is a prerequisite for extending traditional software testing to cover concurrency errors.

8. Related work

There are many research results that partly overlap with our work, and solve some of the problems we address, but under restricted conditions or in a limited scope. The main difference between our work and earlier approaches is that we aim to address a larger set of fundamental problems at once, without placing strict requirements and assumptions on the software being studied.

There have been other complete system simulators than Simics, similar in design: SimOS [23] and a PDP-11 simulator [7]. Various research experiments that involve SimOS and Simics illustrate the benefits of complete system simulation for understanding software behaviour [2, 13]. Rosenblum et al. presented annotations, a method of connecting scripts to events in the software under study [22]. Simics,

as shipped by the vendor, has limited support for debugging applications with GDB, which allows interactive debugging under favourable circumstances.

Virtualisation programs, such as Xen [8], are generally not isolated from indeterministic input from the outside world, and can therefore not provide deterministic execution. Pervasive debugging, a method for deterministic debugging of distributed applications running in Xen domains [15], can be regarded as a light-weight version of a subset of holistic debugging. It provides more limited observability and only supports distributed systems with homogeneous hardware. Furthermore, a pervasive debugger is focused on interactive debugging, and provides no support for automation. It does not include an abstraction translation stack, and instead requires modifications to the virtual machine of each process being observed [14].

Deterministic replay is a popular research technique for reproducing intermittent errors [19, 25]. Replay tools typically require modifications, either to run-time system, operating system, or hardware, and generally require a homogeneous environment.

There is a multitude of other tools, too many to enumerate, for modifying distributed programs, run-time systems, or hardware to trace behaviour for debugging or performance analysis. They do not achieve reproducible experiments, however. There are literature surveys on techniques and problems related to tracing and debugging concurrent programs, covering many such tools [16, 21, 24].

There are also some tools that attempt to raise the debugging abstraction level, either by understanding high-level abstractions, or allowing the user to program the debugger [5, 12].

9. Conclusions

We present holistic debugging, a robust platform for building new, scalable and robust tools that can inspect all state in a distributed software system, and that does not suffer from probe effect and nonrepeatability. We show how a holistic debugger can be constructed from a complete system simulator and stacked abstraction translators that map low-level simulator data to high-level program information. We present the debugger shepherd concept, a programmable and extensible environment that can observe distributed state in the system. We have created Nornir, a prototype holistic debugger implementation, built with the complete system simulator Simics, the GNU debugger, a virtual machine translator for Linux, and a debugger shepherd implementation.

Holistic debugging is a new method for building tools. It has some major benefits, but also disadvantages, some of which may be inherent, and others addressed with time, if holistic debugging becomes an accepted method. The set

of benefits and disadvantages, however, is radically different from existing methods. Unlike alternative approaches, holistic debugging assumes very few properties of the software being inspected. We believe that it is an important key for solving several problems that the computer science community have researched for decades, without producing widely accepted solutions, for example distributed system debugging, testing for concurrency errors, and performance profiling for distributed systems. We believe that some day, it will be natural for software developers to test software in simulated computers, write debugging and white-box test routines that are reused by their colleagues and customers, use chaotic timing and fault-injection models that stress their software more than can be done today, and that these factors enable them to build larger software products of higher quality than is possible today. That day is still far into the future, but our work is a step in that direction.

Acknowledgements

This work is funded by the ARTES research initiative, Vinnova, and the EU FP6 project RUNES. We would also like to thank David Larson, who created the first Verdandi implementation, Anders Wallberg for contributions to Nornir, Erik Hagersten for supervision, and Virtutech for Simics support and licences.

References

- [1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.
- [2] A. Alameldeen, C. Mauer, M. Xu, P. Harper, M. Martin, D. Sorin, M. Hill, and D. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *5th Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2002.
- [3] L. Albertsson. Temporal debugging and profiling of multimedia applications. In *Multimedia Computing and Networking 2002*, Proceedings of SPIE, pages 196–207, Jan. 2002.
- [4] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, 2004.
- [5] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1–31, 1995.
- [6] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using runtime paths for macro analysis, 2003.
- [7] J. K. Doyle and K. I. Mandelberg. A portable PDP-11 simulator. *Software Practice and Experience*, 14(11):1047–1059, Nov. 1984.
- [8] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [9] M. Ekman. *Strategies to Reduce Energy and Resources in Chip Multiprocessor Systems*. PhD thesis, Chalmers University of Technology, Dec. 2004.
- [10] J. Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.
- [11] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, and J. Hennessy. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58. ACM, Nov. 2000.
- [12] M. Golan and D. R. Hanson. DUEL - a very high-level debugging language. In *USENIX Winter*, pages 107–118, 1993.
- [13] S. A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, Feb. 1998.
- [14] A. Ho. Personal communication, 2005.
- [15] 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)*, Nov. 2004.
- [16] J. Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.
- [17] T. C. Jones. *Estimating software costs*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.
- [18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [19] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.
- [20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, Feb. 2000.
- [21] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.
- [22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.
- [23] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.
- [24] W. Schutz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [25] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology (KTH), May 2000.