Quarantine: Fault Tolerance for Concurrent Servers with Data-Driven Selective Isolation

Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin, Madison

Abstract

We present Quarantine, a system that enables data-driven selective isolation within concurrent server applications. Instead of constructing arbitrary isolation boundaries between components, Quarantine collects data to learn where such boundaries should be placed, and then instantiates said barriers to improve reliability. We present the case for data-driven selective isolation, and discuss the challenges in realizing such a system.

1 Introduction

Commodity CPUs are now parallel processors. Parallelism has become a tenet in most processor designs, with some researchers suggesting that the era of many-core systems consisting of thousands of cores is fast approaching. As this trend continues, new software systems are being written for exploiting this parallelism, while operating systems are being redesigned to both accommodate parallel applications and take advantage of the underlying architecture [2, 3].

While this advent of parallel systems is likely to be pervasive, it will have an especially large impact on server hardware. Cloud computing datacenters, another ubiquitous entity in today's virtual world, rely heavily on parallel hardware [1]. Single-installation server machines follow suit, with a typical x86-based server containing at least eight CPUs [1]. On the other hand, most server software, such as Apache, PostgreSQL, and sendmail, follow a thread-per-request architecture; the industry is continually trying to improve hardware to support these applications [20].

Though concurrency is both a natural part of servers and an important factor in improving their performance, writing production-level concurrent software is generally harder than writing sequential programs. Debugging multithreaded software is hard [17]; atomicity violations, data races, and other forms of concurrency bugs can be both hard to find during testing, and can be tough to reproduce. This inherent difficulty in writing concurrent programs has inspired a spur of research on detecting concurrency bugs, either dynamically [5, 6, 13, 17], or by using static methods [9, 21].

The construction of reliable, concurrent servers, however, remains a critical challenge. The problem is that detection of concurrency bugs alone does not guarantee reliable software; it is also necessary to recover from faults caused by unexpected bugs. In other words, concurrent server software needs to be fault tolerant.

Conventionally, whenever fault tolerance is a concern, fault-isolating *boundaries* are provided between isolated components in the system, and failing components are automatically recovered. This pattern can be seen in the design of single-machine operating systems, which provide hardware-based memory isolation boundaries between non-trusting processes. In a more general sense, distributed systems are designed such that each node can be recovered individually, while research ideas such as Rx [14] modify this concept slightly so that time, instead of components, is divided. Specialized methodologies are used to draw more fine-grained boundaries; for example, Nooks [18] draws boundaries between the main kernel and drivers in an OS.

If boundaries can be used to achieve robustness, why aren't they sufficient enough for concurrent server software systems? Unfortunately, most systems create *static* boundaries between subsystems based on developer *intuition*; this has been the norm in approaches to date. Thus, boundaries are likely to be misplaced; isolation between components has traditionally been expensive [22], and concurrent software systems benefit only if the isolation is sufficiently fine-grained to find and recover from concurrency bugs.

In this paper, we introduce *Quarantine,* a framework for building robust concurrent server-based applications with *data-driven selective isolation*, i.e., isolation that is only instantiated when it is needed between the com-

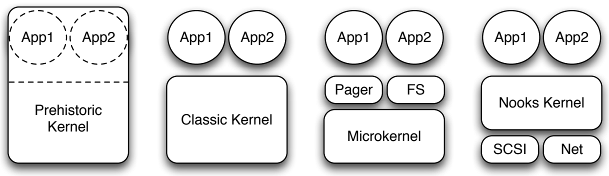

Figure 1: **Kernel Architectures.** *The figure represents the different kernel protection architectures that have evolved over time. In the prehistoric kernel, there is no isolation between components of the system (i.e., the OS is just a library); in the classic kernel (e.g., a typical modern system), there is a fence between the kernel and each process; microkernels place high-level pieces of the kernel (e.g., VM pagers, file systems) into their own ; finally, Nooks organizes device drivers in their own protection domains.*

ponents. There are two key pieces in Quarantine. The first is the Quarantine Programming Model (QPM), a programming model that supports flexible isolation between the modules. This framework enables developers to write server-class concurrent applications in a familiar but somewhat stylized form; a Quarantine compiler then converts said code into its final form with barriers between isolated components as specified. The second is a long-term data-analysis engine, the Quarantine Isolation System (QIS), which collects and analyzes data from deployed servers in order to determine which modules of a server are faulty and in need of isolation. The analysis engine tries different combinations in the field, as well as studies crash dumps, to determine where best to place boundaries. These two pieces combine to enable Quarantine determine where boundaries should be built, and erect them in deployed systems, while incurring isolation overhead only for necessary boundaries.

The rest of this paper is organized as follows. In Section 2, we discuss how boundaries have been used in the past. We then present the Quarantine system in Section 3, discuss some ideas in Section 5 and conclude in Section 6.

2 Background

We now present background relevant to Quarantine. We first explain how process boundaries have been used traditionally to achieve fault tolerance towards concurrency bugs and then examine process boundaries in detail. Finally, we discuss why process boundaries are not currently sufficient for tolerating concurrency bugs.

2.1 Fault Tolerance for Concurrency Bugs: The Current State

While there has been ample research into unique fault tolerance methodologies for various systems, application-level crash and restart is the practically preferred method for faults from concurrency bugs in single-node systems. One reason for this is that concurrency bugs are mostly non-deterministic [8, 12, 23] and hence cause transient faults; rebooting or restarting is a well known method for recovering from such faults [4, 11]. Another reason is that most of the existing studies on concurrency bug identification techniques [5, 6, 13, 17] are optimized towards helping *programmers* identify bugs rather than for fault tolerance: they produce false positives and concentrate on only specific subclasses of bugs. As a result, process-level isolation techniques are currently the most used paradigm for concurrency bug fault tolerance in server software.

2.2 Process Boundaries

Process-level isolation is usually provided by the operating system, and is based on hardware-based protection of address space boundaries. In general, failures can manifest themselves in different ways [11]; address space boundaries work on the idea that faults might ultimately cause a violation of the boundary before any real damage is done. There is a higher chance of the violation happening if the boundaries are fine-grained, and hence, finer-grained boundaries have been introduced as systems have evolved. One example of this trend is found within the operating system: Figure 1 shows the evolution over time, from prehistoric systems with no protection domains, to classic systems that separate the OS from user applications, to more modern approaches such as microkernels [7, 15, 16, 24] and Nooks [18].

As is obvious from the diagram, these systems all place boundaries between components in an entirely static manner. For example, Chorus [16] and Mach [15] are designed to push certain pieces of the kernel into user space, with the goal of enabling users or developers to tailor said components for specific workloads. Thus, the presence of user-level pagers or file systems defines the new protection boundaries in the system. Nooks makes similar decisions about protection boundaries, focusing instead on device drivers [18].

2.3 Why Extremely Fine-Grained Protection Doesn't Work

Clearly time has shown that isolation boundaries are useful; though debate exists as to how many boundaries are needed, there is little doubt that isolation is of great utility. Even limited embedded environments now regularly include hardware-based protection mechanisms. Thus, if isolation is useful, why not then push the idea further, isolating components as small as possible so that most concurrency faults can be detected?

There is one primary answer to this question: performance. As a simple demonstration of the costs of hardware-based protection, we measured the cost of making a function call within the same protection domain and compared it to the cost of making a "remote" procedure call into a different process on the same machine. As expected, while the procedure call only takes a few cycles, the cost of the inter-process communication is proportional to the cost of two context switches (about 4 to 6 microseconds on a modern machine), or roughly three orders of magnitude worse.

It is thus clear that isolation boundaries should only be constructed where absolutely necessary. Needless protection crossings can greatly harm performance with little added benefit.

2.4 Other Protection Mechanisms

We have previously explained how address space isolation has been useful, and why it has been found to be more useful than specialized bug-detection mechanisms [5, 6, 13, 17]. However, what about alternative approaches? For example, languages with strong type systems have lower overhead and higher fault detection abilities.

One reason language-based protection does not suffice is that these mechanisms usually apply to higher-level languages. However, many important concurrent server systems, such as Apache, PostgreSQL, and sendmail, are still developed in low-level languages like C or C++. Further, a recent study of open-source projects revealed that C accounts for roughly 40% of open-source project code [10]. Because of the efficiency, control, and large installed-bases low-level languages provide, they are likely to be an important part of infrastructure for years to come.

2.5 Summary

In short, we can learn the following from the current techniques and previous explorations into fault tolerance mechanisms for concurrency bugs:

- Isolation in general, and hardware-based address space isolation in particular, are useful as fault-detection techniques for fault tolerance.
- In current systems, fine-grained isolation is better for reliability, but worse for performance.

Thus, we need something that allows us to build isolation boundaries only at optimal places, to isolate potential bugs while minimizing the impact on performance.

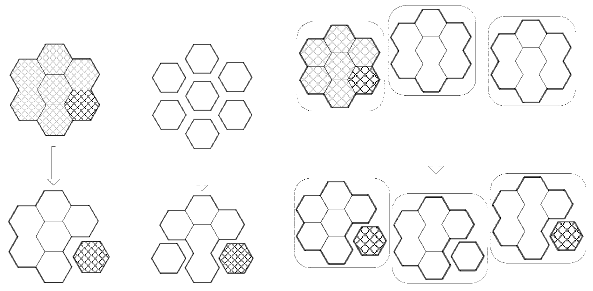

Figure 2: **Sensible-boundary identification mechanisms.** *The figure illustrates some methodologies that can be used to identify the optimal boundaries to enforce; the concurrent server is imagined to be composed of separable hexagons, and the shaded hexagons represent faulting components. In the optimistic approach, the system is allowed to fault once and the culprit component is isolated in future; in the pessimistic approach, all components are isolated at first and the trusted components are later combined; finally, the cooperative approach collects information from multiple systems to decide on what to isolate.*

3 Quarantine

The goal of Quarantine is to achieve fault tolerance (for concurrent server software) with both good reliability and little performance overhead. In order to do this, Quarantine isolates only parts that it believes are faulty using address-space boundaries. There are two requirements for achieving this:

- Splitting a concurrent server into fine-grained components, each of which may be individually fault-isolated.
- Deciding which of these components should be isolated during run-time in order to improve the overall reliability of the system while maintaining reasonable performance overhead.

We decompose Quarantine into two parts corresponding to these two requirements. A straightforward way to achieve the first requirement is to allow the programmer to indicate separable components within the system; the first part of Quarantine is thus a stylized C programming model to enable the system to determine clean and separable boundaries within the server. The second part of Quarantine is a set of techniques to identify an optimal set of boundaries, balancing both performance and reliability of the system.

3.1 Quarantine Programming Model

The purpose of the Quarantine Programming Model (QPM), a set of language modifications for C, is to allow the programmer to divide the system into separable components that can be isolated if necessary. This goal can be achieved in two steps:

- Provide an easy, intuitive way of identifying fine-grained separable components.

- Associate a well-defined, non-overlapping, run-time memory boundary with each component, as required for address-space isolation.

While designing the modifications, we should neither reduce the flexibility of C, nor introduce any performance overhead when no components are being isolated. In addition, QPM should resemble the original C programming model as much as possible.

For the first step, we can divide along arbitrary sections of code, functions, files or any combination of these. It is best to align the separable component boundaries to function boundaries, since function boundaries are both easy to understand for the programmer and are sufficiently fine-grained in most systems.

The next step is to define run-time memory boundaries for a separable function group. This step is challenging because, in the presence of pointers, C does not define any semantic to associate functions with memory areas (that are accessed by code belonging to the considered function during run-time).

To deal with pointers, we need to modify the C programming model to define the memory boundary. We devise a “inout copy” semantic for pointers; whenever a pointer is one of the parameters of a function call, and the function call is to a separate isolated component, the “data referenced to” by the pointer is copied to the isolated component, and copied back when the function returns. The same semantic applies if a pointer is a part of a parameter (in the case of a structure or union parameter). Thus, if components are isolated, pointer references are also handled as if they are a part of the stack frame. However, for function calls between non-isolated components, the normal C pointers semantic is obeyed. The “data referenced to” by a pointer is not semantically defined in the low type-safety environment of C (considering unions, pointers and type casting); we purport to introduce new annotations that can be used by the programmer to define this for the necessary pointers. Global variables can be handled in a similar manner.

3.2 Quarantine Isolation Subsystem

To achieve both good performance and reliable fault tolerance, the Quarantine Isolation Subsystem (QIS) performs data-driven isolation: it analyzes crash reports and isolation overheads, using the data to instantiate boundaries at optimal places within the system. We first describe, in a broad sense, some approaches to determine which places are optimal. The methods described here are illustrated in Figure 2.

Pessimistic approach: The system is first run with all components isolated from one another. The QIS observes the components for some period of time, or until a specified number of function calls happen to it; the isolation boundaries are removed gradually as QIS begins to trust the components. One disadvantage of this approach is that the system will run slowly during the initial phase when none of the components are trusted. However, assuming that the faulty components perform address space violations during the initial phase, this method results in high reliability.

Optimistic approach: The system is run without any boundaries initially. When any crash occurs, a list of components that might have caused the fault is determined (for example, by analyzing the stack trace), and those components are isolated during the next run. The isolated components can be combined afterwards if their behavior seems to be trustworthy. Although this method has zero run-time overhead (if there are no faulty components), each faulty component would cause a whole system failure at least once. In addition, reliability is notably affected by the correctness of the faulty-component-determination method used.

Cooperative approach: Cooperative Crug Isolation (CCI) [8] is a project which automatically diagnoses deployed software failures by collecting information from multiple installations. A similar method can be used for Quarantine; this method would be especially useful with systems in broad deployment. Such a method, though complex to implement, would isolate suspicious components alone without faulting even once.

QIS uses a combination of these approaches: it records the frequency of faults corresponding to each component, the time to recover, and the overall performance impact of the isolation boundaries in order to intelligently determine which components to isolate. For example, QIS might determine from its analysis that it is best to decompose a 100-component system into 10 big isolated parts initially; as time progresses, it might further decompose one of those ten, while combining the other nine. QIS thus ensures that the balance between performance overhead and reliability is optimal.

4 Discussion

In this section, we list some thoughts about the purpose and design of Quarantine.

Is Quarantine limited to concurrency bugs? Quarantine has been designed with concurrent server systems in mind. Concurrency bugs have two characteristics that are inherent in the motivation of Quarantine and have been exploited in its design: concurrency bugs are hard to detect before deployment, and after deployment, most concurrency failures can be solved by restarting. However, the basic concept of data-driven, selective-isolation can be applied to a wider variety of faults and systems.

Does Quarantine require new systems to be built from scratch? The QPM necessitates programmer involvement in converting an existing system into a Quarantinable-system; although the "inout copy" semantic introduces only little programming nuisance for sequential programs, for a concurrent program, the semantic introduces a notable difference in the programming model; two concurrent function calls might actually be working on two different copies even if the functions are given pointers to the same data. However, QPM facilitates gradual conversion of an existing source code base. One way to avoid this entirely would be to look into automatic component identification techniques.

Are address space fault-identification techniques sufficient? Programmer defined asserts, specialized fault identification techniques [5, 6, 13, 17], stronger type safety mechanisms and other methodologies can enable quicker identification of a wider spectrum of failures. Address space isolation, as used in Quarantine, provides a basic framework; since the QIS separates out suspected components into separate address spaces, it should be trivial to apply any additional fault detection technique on the isolated component.

5 Current State

We have a prototype implementation of the Quarantine framework, in which separated parts are automatically restarted when a fault happens. During restart, any requests which were not replied to by the module are replayed, making the restart transparent to the other modules. Replay works by logging function calls (along with the parameters) and returns to the main memory. One major challenge was copying the data pointed to by a pointer, which was solved with minimal programmer annotations. The prototype implementation works for transient faults which do not propogate beyond each isolated component, if the isolated components are designed such that they can be restarted. It remains to be seen whether these conditions are suitable when building large software systems, and how fault detection and restart can be extended to be more effective.

6 Conclusions

Just as John Snow used data to determine the likely source of a Cholera outbreak [19], so too should robust systems. With Quarantine, such data is used to decide where isolation boundaries should be placed. We hope that Quarantine, with its modified programming environment and data-analysis based fault tolerance, will enable a new generation of robust concurrent services to be built. Further, we believe that the idea behind data-driven isolation is a broad one, and thus encourage experimentation with similar empirical approaches to reliability. Wherever a service is deployed widely, different approaches can be employed, and data can be used to gauge the effectiveness of alternatives. In doing so, perhaps a new era in the construction of reliable systems will be realized.

Acknowledgments

We thank the anonymous reviewers for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank L. Arulraj and S. Panneerselvam for their comments on earlier drafts of the paper.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0811657, CNS-0834392, CCF-0937959, CSR-1017518, as well as by generous donations from NetApp and Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or other institutions.

References

[1] BARROSO, L. A., DEAN, J., AND HLZLE, U. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro 23*, 2 (March 2003), 22–28.

[2] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)* (San Diego, California, Dec. 2008).

[3] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Vancouver, BC, Canada, Oct. 2010).

[4] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)* (San Francisco, California, Dec. 2004), pp. 31–44.

[5] ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., AND OLYNYK, K. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Vancouver, BC, Canada, Oct. 2010).

[6] FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming 71*, 2 (Apr. 2008), 89–109.

[7] HANSEN, P. B. The nucleus of a multiprogramming system. *Commun. ACM 13*, 4 (Apr. 1970), 238–241.

[8] JIN, G., THAKUR, A., LIBLIT, B., AND LU, S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10)* (Reno/Tahoe, Nevada, USA, October 2010).

[9] KAHLON, V., YANG, Y., SANKARANARAYANAN, S., AND GUPTA, A. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th international conference on Computer aided verification (CAV '07)* (Berlin, Germany, July 2007).

[10] KERNER, S. M. Most Popular Open Source Languages. http://itmanagement.earthweb.com/osrc/article.php/3834596/Most-Popular-Open-Source-Languages.htm, 2009.

[11] KIENZLE, J. Software fault tolerance: An overview. In *Reliable Software Technologies Ada-Europe 2003*, vol. 2655 of *Lecture Notes in Computer Science*. 2003, pp. 641–641.

[12] LU, S. *Understanding, Detecting and Exposing Concurrency Bugs*. PhD thesis, University of Illinois at Urbana-Champaign, 2008.

[13] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)* (San Jose, California, Oct. 2006).

[14] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)* (Brighton, United Kingdom, October 2005).

[15] RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* (Palo Alto, California, 1991), pp. 31–39.

[16] ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LANGLOIS, S., LEONARD, P., AND NEUHAUSER, W. CHORUS Distributed Operating System. *Computing Systems 1*, 4 (1988), 305–370.

[17] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 391–411.

[18] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003).

[19] TUFTE, E. *The Visual Display of Quantitative Data*. Graphics Press, 2001.

[20] VEAL, B., AND FOONG, A. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07)* (Orlando, Florida, USA, Dec. 2007).

[21] VOUNG, J. W., JHALA, R., AND LERNER, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE '07)* (Dubrovnik, Croatia, Sept. 2007).

[22] WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)* (Asheville, North Carolina, Dec. 1993), pp. 203–216.

[23] WEERATUNGE, D., ZHANG, X., SUMNER, W. N., AND JAGANNATHAN, S. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)* (Trento, Italy, July 2010).

[24] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., AND BARON, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)* (Austin, Texas, November 1987), pp. 63–76.