Safe at Any Speed: Fast, Safe Parallelism in Servers

John Jannotti Kiran Pamnany
Brown University
{jj,kiran}@cs.brown.edu

Abstract

Many applications take advantage of parallelism to increase performance. Servers are a particularly common case as they must multiplex resources across many simultaneous users. Unfortunately, writing concurrent applications is difficult and prone to subtle and non-deterministic bugs that are difficult to reproduce. We advocate an approach to developing concurrent programs that is *safe by default*. Conservative static analysis determines when two code segments may safely run in parallel, and a runtime scheduler respects these constraints.

We have built an analyzer for event-driven servers that discovers data sharing to find safe parallelism among event handlers. As a prototype, the analysis currently considers only global data, assuming that request-specific data structures passed to event handlers are completely independent. We have also begun work on a runtime system that schedules event handler execution within the constraints determined by the analyzer. For performance reasons, the scheduler makes additional conservative assumptions about contention.

We have analyzed thttpd, an event-driven web server. We show how our system can be used to increase performance without complex synchronization schemes.

1 Introduction

The usual approach to building concurrent servers is to start by thinking about the serial version of request handling and then consider how concurrent executions of the serial code might cause race conditions. Each potential race must be recognized and appropriately synchronized before the server may be executed safely on concurrent requests. This effort represents a substantial intellectual investment for even moderately complex servers.

Yet servers must take advantage of concurrency in order to support their client loads. We believe that a conservative static analysis can be used to help provide safe, exploitable parallelism. In combination with a runtime system that schedules according to constraints determined during analysis, developers may be assured that two segments of code that potentially conflict are never run concurrently. As such, code developed for serial execution may be executed safely in this environment — it is *safe by default*. Since this analysis must be conservative, it may exclude some potential concurrency. The analysis will explain its discoveries so that developers can modify their code to enable more parallelism.

The key advantage to default safety is that developers start with a correct application and apply development effort until they are satisfied with performance. Today, developers start with incorrect code and must apply development effort until they fix *all* races. Tomorrow's developers might not bother to expose every bit of potential parallelism once they are satisfied with performance, but they will nonetheless have a correct application.

This paper outlines our vision for safe servers, and then describes the analysis and runtime system necessary to run these applications safely and efficiently. It also discusses our initial experiments to verify that the necessary components can be built. We have developed an analysis tool for determining the independence of handlers in event-driven programs, and used it to analyze thttpd, an existing event-driven web server. We have also created a runtime scheduler that executes event handlers in multiple threads while respecting constraints on their concurrent execution.

2 A Vision for Safe, Fast Servers

We envision a significantly improved development model for concurrent applications, particularly

servers. Today's most common architecture for concurrent servers uses a thread per request. While the code for individual threads is similar to the code written to handle a single request in a sequential server, it is the subtle set of required changes that causes problems. Locks or other synchronization primitives must be used to protect access to shared resources, but the use of these primitives introduces the possibility of deadlocks, livelock, potential starvation and other difficulties.

We liken the current state of concurrency management to the state of memory management without garbage collection. Using explicit memory management, programmers are expected to avoid memory leaks through a combination of discipline, intelligence, and careful debugging. Garbage collection eliminates the problem entirely, and incidentally results in simpler interfaces. Similarly, today's developers must explicitly manage the more difficult problem of concurrency, often leading to brittle interfaces that must be invoked under specific conditions with respect to any number of locks. Although some tools exist, both static [6, 7] and dynamic [10, 11, 12], to help track down concurrency bugs, we aim to eliminate the source of these bugs.

Since we aim to help developers exploit parallelism safely, prior approaches that have attempted to automatically find parallelism [3, 4, 5] might seem similar. The key difference in approach is that we believe developers are quite good at finding parallelism (especially in servers that appear quite parallel), but they sometimes make mistakes. An error that introduces unsafe parallelism leads to an incorrect program, while an error that hides potential parallelism is simply a performance problem. Therefore tools should focus on eliminating unsafe parallelism, rather than finding more safe parallelism while unsafe parallelism goes undiagnosed.

Programming model We are inspired by the event-driven programming style for managing concurrency in which a main loop receives events from the operating system and invokes *handlers* to process these events. These handlers are invoked with callback arguments which we refer to as *contexts*, which allow the handler to determine details about the work that must be performed. Handlers invoke asynchronous services and register further handlers to react to service responses. There is debate [1] over the ease of programming of event-driven systems, but they have an important correctness advantage over threaded systems: they naturally provide atomicity [8] for the code in each handler. Each handler is guaranteed to run to completion without interference from other handlers.

Unfortunately, a serial event loop precludes any parallelism between event handlers. In order to exploit multi-processors, a new execution model for event-driven applications has been described [14] in which programmers manually specify *colors* for event handlers to explicitly enable parallelism. Event handlers of a given color never run concurrently with other handlers of the same color.

We believe developers should program to an atomic model, whether event-based or thread-based. It is the job of the static analysis and runtime system to ensure atomicity despite running much of the code in parallel. In the near-term, it is easiest to imagine implementing this vision for programs that are written in an event-driven style, or use short, run-to-completion threads. For programs in this style, when the analysis errs on the side of caution and prevents parallel execution while a particular handler is running, the slow-down is limited to the lifetime of the handler. We believe our approach can eventually be used with programs written in a long-lived threaded style as well, however programmers will need to indicate the segments over which they require atomicity to be preserved.

Scheduling model An atomic programming model allows developers to reason about the correctness of their programs, but they must consider the scheduling model in order to understand performance. The runtime will guarantee that atomic segments that might interfere with one another do not run in parallel. Therefore, the constraints discovered by the analyzer will be of interest to the programmer; given the source of a conflict, she might work to remove the constraint. For example, the conflict may be the result of a rarely taken error handling path that increments a global statistics counter. Here, a lock may be inserted around access to the the counter, which would allow the analyzer to remove the constraint. An explicit lock over a few lines of code would allow more parallelism than a constraint that would apply to an entire handler, regardless of whether the error handling path was taken.

Alternatively, the programmer may split the segment into several individually atomic stages. Figure 1 illustrates a well partioned example in which the work required to process a request is split into

six functions. After each connection is accepted, a chain of handlers processes the request. In this case, most handlers are independent across requests, so they may execute in parallel. The Query and Log functions access global variables, however, so they are constrained from running concurrently. If the server were broken into fewer stages, for example by handling the database query and response preparation together, the analysis and scheduler would be forced to avoid concurrency for response preparation in separate requests.

3 Approach

We have advocated a programming model in which code segments are implicitly prevented from concurrent execution if they share data unsafely. The programmer may then spend time explicitly removing these constraints to increase performance. This section describes the techniques used to find and enforce these constraints.

3.1 Analysis

The analysis must determine when the set of data read by one handler overlaps with the set written by another. For safety, this analysis must be conservative. A handler that *might* access a particular data object must be treated as though it will.

Independence Consider two potential handlers, f and g. At runtime, the scheduler will consider specific *invocations* of handlers on specific context arguments, for example $f(c_f)$ and $g(c_g)$. The static analysis must operate on handlers, not invocations, but should record information that allows the runtime scheduler to make decisions about invocations.

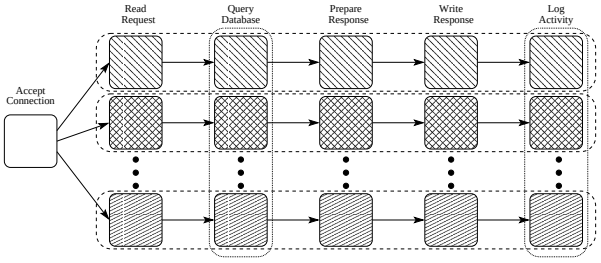

Figure 1: A request is processed by six handlers. The handlers are analyzed to obtain safe scheduling constraints. Grouped handlers have been deemed incompatible for concurrent execution. Vertical groups are contending on global variables.

It would be infeasible to trace reachability through contexts at runtime to determine potential conflicts.

A invocation possesses two *roots* from which all accesses must occur — the global variables of the program and the context it is invoked on. Although the analysis cannot observe the specific context for a particular invocation, it may consider an abstract representation of the context to determine a conservative estimate of reachable objects. If the objects reachable by f on any given context, and g on any given context are disjoint, then we know that $f(c_f)$ may be scheduled simultaneously with $g(c_g)$ without conflict for any given c_f and c_g. The analysis is performed at compile-time and thus cannot take into account the specific arguments to f and g, nor may it consider the state of the program's global variables.

If two handlers conflict because of their use of a global, they are unsafe to execute simultaneously under any circumstances. Conflicts on contexts are more subtle. Context are likely to be self-contained and unreachable from other roots. When analysis determines this to be true of the contexts in a particular program, it makes sense not only to ask, "Are f and g independent when invoked on unknown contexts?" but also "Are f and g independent when invoked on contexts that are known to be different?" Since we expect many handlers to conflict only when invoked on the same context, we believe that differentiating these cases is critical to uncovering significant parallelism.

The results of the analysis must be reported to the programmer to aid incremental improvement, as well as compiled into the application for use by the scheduler which must decide at run-time whether f may be executed with this *particular* c_f while g is simultaneously executing with a particular c_g.

Multiple contexts The analysis may be generalized by considering several independent contexts. For example, a program may use a context type which contains two kinds of data: the identity of the client making a request, and the details of the request itself. There may be event handlers that conflict if the identity of the client is the same, but do not conflict if the request details are the same. If the analysis considers only the single unified context, it will forbid all concurrent execution of these handlers, even when their contexts differ, because those contexts might share the same client. This is

overly conservative. In such cases, the static analysis and scheduler might be directed to consider client equality independently.

OS Resources The analysis must also consider how the program uses resources such as files and sockets. The static analysis we have described assumes that conflicting objects are blocks of memory, but they may also be operating system abstractions. Thus, the analyzer must be aware of the semantics of the system calls that pertain to these abstractions, and recognize safe concurrency in handlers that access objects of these types. Two handlers that read and write the same file may not be executed concurrently.

3.2 Scheduling

The runtime system must honor the constraints identified by the analysis. Handlers may only be invoked if the analysis shows that they are safe to run simultaneously with all other invocations currently executing. A naïve implementation of such a scheduler might have each thread that is ready to invoke a new handler query all other threads in order to determine the handlers they are executing. The thread would then select a non-conflicting handler to execute from a pool of pending invocations. However, such an approach would exhibit very poor performance (assuming short-running invocations) due to the need to synchronize access to this shared state before every invocation.

Work in thread scheduling for multiprocessors [2, 13] may improve this inefficiency; however, we have chosen to begin with a simple conservative approximation (described in the next section) in an effort to experimentally validate our analysis.

4 Validation

We have begun work by building an analysis tool using CIL [9], a C language analysis framework, and a scheduler using ideas from libasync-smp [14].

Analysis In our proof-of-concept analysis tool, we make several simplifying assumptions. First, we only analyze global data. We assume that *any* two handlers will conflict if passed the same context. This is a conservative assumption until our analysis is improved. On the other hand, we assume that handlers do not conflict, except through global variables, when their contexts are different. This is *not* a conservative assumption but rather a pragmatic one. In the programs we have built and examined, it is common for contexts to be built out of completely unshared data. Pragmatic or not, this assumption must be removed when our analysis matures.. Finally, we have not begun to analyze explicit locks or OS primitives, such as file access.

Our analyzer begins by identifying the event handlers. For each handler, a complete call graph is generated, and the set of globals read and the set written are determined for each function called. When accessing a global variable, substructure is distinguished so that a write to `stats->num_connects` is not conflated with `stats->num_users`. Array indices are conflated, however. No attempt is made to distinguish `users[1]` from `users[2]`.

Currently, the analyzer performs rudimentary alias analysis in order to identify code that passes globals to functions where they are read or written. In some cases these aliases are tracked precisely, in others the analysis conservatively assumes the worst.

Combining the call graph and the global access sets, the analysis determines the set of globals read and the set written for each handler and all its descendants in the call graph. Conflicts are reported for any two handlers in which one handler reads a global and the other writes it.

Scheduling We have built a scheduler that conservatively respects scheduling constraints without requiring a global lock. Our scheduler extends ideas described in libasync-smp [14] to schedule our richer constraints. In libasync-smp, a handler is labeled by a *color*; like-colored handlers may not be executed simultaneously.

A simple color based scheduler is insufficient to express the constraints of the program in Figure 1 in which a handler like Log may not execute with another handler for the same request *or* another Log handler, but it would undesirable to use the same color for all such handlers as that would preclude safe concurrency. To address this limitation, we introduce a second level of color, called a *hue*. Our scheduler ensures that no two handlers of the same hue or of the same color execute concurrently. Handlers may also have no hue or color; these can be

executed without regard to the hue or color of other handlers. The scheduler uses a queue per color and a queue per hue to avoid a global lock.

The hue/color scheduler can be used to execute a conservative approximation of the constraints determined by analysis. Although hues and colors follow the same rules, they are used differently to encode the constraints determined during analysis. If two handlers may not be run in parallel, regardless of context, they are assigned the same hue. This implies that when f conflicts with g, and g conflicts with h, they must all receive the same hue—the hue represents the transitive closure of conflict on globals. This rule is conservative. g and h may not really conflict, but this simplification is easy to implement without expensive run-time checks.

However, if two may be run in parallel as long as their contexts differ, they are assigned the same color, not hue. Since we currently assume that *all* handlers with the same context conflict, we use the address of the context structure as the color. Handlers that can be executed in parallel with themselves, even on the same context, can be given no color though we expect this to be rare.

thttpd We have analyzed thttpd, an event-driven web server. thttpd was written in a traditional event-driven style. It contains no locks, and tries to avoid blocking in handlers by using non-blocking network I/O. However, it does not use non-blocking file I/O, as doing so remains a complicated proposition. Serial execution combined with blocking in file I/O makes thttpd a poor choice for disk-heavy workloads.

We hoped to learn how difficult it would be for a developer to safely alleviate this problem with the help of our analyzer and multi-threaded scheduler. thttpd performs quite well on cached workloads, so our goal was to prevent disk access from interfering with cached requests. thttpd mostly reads the file system through `mmap` therefore most blocking occurs when memory is paged in while writing to the network. In thttpd, `handle_send` is the event handler that performs this function.

The `handle_send` event handler had to be made independent with itself (when invoked on different contexts) so that multiple read requests could be in flight simultaneously. Our analyzer found six global variables that are read and written in `handle_send`. Five of these are modified in `really_clear_connection` which is called by `handle_send` if the connection must be closed. The sixth is a statistics counter that is updated in `handle_send` itself.

There are two ways to remove such conflicts. One is to use locks to protect these accesses. The analyzer would then, once complete, recognize this protection and no longer report these accesses as conflicts for `handle_send`. The disadvantage of this approach is that if other handlers also access these globals, it would be necessary to use these locks at each access in order to make `handle_send` independent of those handlers as well. Another way to remove conflicts is to simply defer the variable update to a separate handler. This not only makes `handle_send` independent with itself, but with other handlers as well.

We first deferred the invocation of `really_clear_connection` to a separate handler, changing two lines of code and adding a two line function. The analyzer then reported that `handle_send` had a single conflict with itself–the statistics counter. We deferred the counter update to another handler, moving two lines of code to a new function and adding the call to register the action. This change removed all conflicts, not only making `handle_send` independent with itself (when invoked with different contexts), but also with almost all the other handlers.

The changes necessary to enable this independence were minimal. The developer is guided at each step of the way with reports that label conflicts and the references that cause them.

5 Conclusions

The use of concurrency in servers should be safe by default. Using static analysis and an informed scheduler, naïve applications can be made safe from race conditions. If default safety comes at too high of a cost, the analysis can be used to inform developers where contention occurs so that it may be eliminated. Though our experiments are preliminary, we have been encouraged by the degree to which a real-world server, thttpd, could be analyzed and parallelized despite several conservative assumptions.

In the past, static analysis has been used to find likely bugs in programs, but human attention is often required to confirm the bug's existence and fix the problem. One may view the analysis we propose

as a search for concurrency bugs and the scheduler as an automated fix for those bugs. In some cases, these bugs may not really exist, but if dependability is valued more than performance, the preference will be to err on the side of caution and automate the "fix."

References

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proc. USENIX 2002 Annual Technical Conference*, June 2002.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the 10th annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, 1998.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. In *Proceedings of the IEEE*, volume 81, pages 211–243, 1993.

[4] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel Distrib. Technol.*, 2(3):37–47, 1994.

[5] Bryan Chan and Tarek Abdelrahman. Run-time support for the automatic parallelization of Java programs. In *The Journal of Supercomputing*, volume 28, April 2004.

[6] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, October 2003.

[7] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proc. SIGPLAN 2000 Conference (PLDI)*, pages 219–232, 2000.

[8] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. SIGPLAN 2003 Conference (PLDI)*, pages 338–349, 2003.

[9] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, 2002.

[10] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2003.

[11] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Computer Systems*, 15(4):391–411, 1997.

[12] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *ACM SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.

[13] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proc. ACM SIGMETRICS Conference*, May 1990.

[14] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proc. USENIX 2003 Annual Technical Conference*, June 2003.