Abstract of "Safe Parallelism for Servers" by Kiran Pamnany, Ph.D., Brown University, May 2011.

Applications must embrace parallelism in order to increase performance on today's ubiquitous multicore processors. Unfortunately, writing correct parallel applications is notoriously hard, partly because the dominant parallel programming model uses threads with shared state and locks for synchronization, a model that is subject to a variety of subtle bugs that can be hard to reproduce.

This dissertation advocates a programming model that enables the safe and incremental addition of parallelism to an application designed for serial execution. This model is enabled by a system composed of `elyze`, a static program analyzer, and `multivent`, a runtime scheduler. Together, `elyze` and `multivent` ensure that an application's code segments run in parallel only when they may do so safely, and guide the programmer in making changes to increase those opportunities.

The system has been applied to two real world server applications written in C: `thttpd`, a web server, and `tor`, the onion router. `thttpd` shows an improvement in performance of up to 15%. An elaborate (and defective) thread pool mechanism can be removed from `tor` without compromising its performance. In both cases, the static and dynamic analyses performed by `elyze` and `multivent` guide the programmer in enabling significant parallelism and increased performance, without the need for complex reasoning about concurrent behavior.

Safe Parallelism for Servers

by
Kiran Pamnany
B. Com., Bangalore University, 1994
Sc. M., Fordham University, 2004

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2011

© Copyright 2011 by Kiran Pamnany

This dissertation by Kiran Pamnany is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ________________________________
John Jannotti, Director

Recommended to the Graduate Council

Date ______________ ________________________________
Shriram Krishnamurthi, Reader

Date ______________ ________________________________
Maurice Herlihy, Reader

Approved by the Graduate Council

Date ______________ ________________________________
Peter M. Weber
Dean of the Graduate School

Vita

Kiran Pamnany was born on March 31, 1974 in Bangalore, India. An undergraduate education in Business Economics did not prevent him from designing and writing software for a living for over ten years. He was then bitten by the academic bug and returned to school in 2005 to earn a doctorate in Computer Science. His post-Ph.D. plans include more time with his family, travel, and a parallel computing research lab.

Acknowledgments

I have many people to thank for helping make this dissertation possible, beginning with my advisor, John Jannotti, for his suggestions, insight, perception and critical eye. JJ shaped the engineer into the scientist. Shriram Krishnamurthi has been a friendly ear and source of invaluable advice for all the many years I have known him. Maurice Herlihy is an inspiration to everyone who has met him.

Dr. Damian Lyons made my return to academia possible. Vivek Goyal made my career possible.

Akshay Kadam and Rajesh Raman have been lifelong friends and inspiration.

Most of all, I thank my wife Nitya Shivraman. This is just one part of the magic she's brought into my life.

To Nitya Shivraman.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Single core processor performance has reached the limits of current semiconductor technology. Processor manufacturers, faced by the "power wall", have introduced multicore processors which are increasingly ubiquitous.

As a consequence, parallel programming is no longer the domain of high performance computing alone; all applications must embrace parallelism to maximize performance.

Servers are an egregious example: the emergence of cloud computing and software as a service is moving applications onto servers, where they must take advantage of parallelism in order to concurrently serve large numbers of simultaneous clients.

Unfortunately, parallel programming is notoriously difficult. It is hard to reason correctly about concurrent behavior. Multiple threads executing and accessing data at the same time may suffer data "races", so named because the application's behavior depends on the result of threads racing to a shared object. As operating systems schedule threads non-deterministically, the same application with the same input can show different results on different executions. This makes data races hard to reproduce, understand, and eliminate.

Parallel applications attempt to eliminate data races by using mutual exclusion constructs such as locks, but these are subject to a wide variety of problems in and of themselves—deadlock, convoying, starvation, etc.

The state of concurrency management is reminiscent of the state of memory management prior to garbage collection. Previously, programmers were expected to avoid memory leaks through a combination of discipline and careful debugging. With garbage collection, the problem is eliminated.

Similarly, programmers are now expected to explicitly manage the even more subtle problem of concurrency management. There are a number of tools, both static [10, 12] and dynamic [22, 30, 38], that help track down concurrency bugs, but ideally we would like to eliminate the source of these bugs: programmer error.

1.2 Thesis Statement

It is feasible and useful to use program analysis to inform both the programmer and a runtime scheduler of parallelism constraints in a server application. These techniques guarantee safe concurrency and enable incremental performance gains.

1.3 Design Principles

This work is guided by the principles described in this section.

1.3.1 Default Safety

The usual approach to building a concurrent server application is to start by thinking about the serial version of request handling and then consider how concurrent executions of the serial code might cause errors. Each potential problem 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.

The core thesis of this work is that a conservative static analysis can be used to find safe, exploitable parallelism. This analysis determines constraints on concurrent execution that are used by a runtime system for scheduling. Together, the analysis and runtime system ensure 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*.

1.3.2 Incremental Performance Gains

A 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. An error leads to an incorrect program. With default safety, an error simply hides potential parallelism which is a performance problem, not a correctness issue.

With this approach, segments of code that might interfere with one another will not run concurrently. Then, to improve performance, the programmer must understand the constraints that inhibit parallelism, and work to remove them. To facilitate this, the analysis must provide detailed information on each conflict that it discovers.

Thus, improving performance becomes an iterative process of removing constraints on concurrent execution, as guided by the system.

1.3.3 Existing Code

It is a guiding principle for this work that it must operate on existing, un-modified server applications. Thus, no annotations are required, nor any re-engineering. The programmer must be guided by the system in making any changes to code.

1.4 Document Structure

The remainder of this document is organized as follows: Chapter 2 examines existing parallel programming models and discusses related work. Chapter 3 explores what safe parallelism means and how it may be accomplished. Chapters 4 and 5 detail the two artifacts that constitute the safe parallelism system. Chapter 6 contains the evaluation of the system on two server applications and Chapter 7 considers future work and concludes.

Chapter 2

Background and Related Work

One of the challenges of parallel programming is that a programmer's error can result in unsafe parallelism, which manifests in a variety of subtle problems. But it is not inevitable that a programmer's error must cause a correctness issue; this is a consequence of the limitations of the commonly used parallel programming models.

2.1 Parallel Programming Models

Threads, or processes, are the most common primitive used to express concurrency; this is true for most of the models discussed in this section. The difference, then, is primarily in the synchronization abstractions offered.

2.1.1 Threads with Locks

The essential idea behind locking is simple: allow only one thread at a time to manipulate a given object or set of objects [16]. Unfortunately, the simplicity of the concept conceals some significant complexities when locks are used in large, real-world software.

Large, widely used server applications such as Apache, PostgreSQL, DB2, and others, all use a thread or process per request, with locks to protect shared resources. It is reasonably straightforward to implement request handling logic this way, but attaining correctness while serving concurrent requests requires painstaking effort.

- Every shared resource must be identified.
- Every access to each such resource must be located.
- The correct lock must be acquired, and released.
- Locking granularity must be correctly decided to receive any gain in performance; this decision is not easily changed.

- Simple race freedom is insufficient to guarantee correctness; a stronger non-interference property, atomicity [13], must be satisfied.
- The incorrect use of locks can introduce bugs such as deadlock, livelock, convoying and priority inversion.

Furthermore, bugs relating to parallel execution are hard to reproduce and track down because they may appear intermittently (thread scheduling is non-deterministic across multiple executions).

Considerable research has been done to address these difficulties, finding races in multithreaded applications, and checking for proper lock use [22, 27, 30, 10, 38]. Locks have been studied for decades, but lock-based multi-threaded programming remains a challenging discipline.

McKenney *et al.* [21] provide a more detailed critique of locks, as do others [25, 33].

2.1.2 Transactional Memory

Transactional memory [15] is an alternative way to protect shared data that avoids many of the problems of lock-based programs. The core idea is simple: execute a group of memory operations as a single atomic transaction. Transactions are usually executed optimistically. Thus, with multiple threads running, a number of transactions may execute concurrently. If a transaction conflicts with another running transaction, only one of them will complete while the other will be rolled back and restarted. Conflicts are detected and resolved by the transactional runtime system.

This abstraction simplifies the task of developing correct concurrent programs as it allows a programmer to reason about parallel behavior in a coarse-grained way. The programmer no longer has to worry about problems such as deadlock, the transactional runtime system can make progress guarantees, transactions are composable and scale well.

However, here too are some difficult problems.

- The programmer must determine which groups of accesses must be atomic, *i.e.* identify the transactions. A mistake may result in a race condition.
- The use of optimistic concurrency creates problems for irreversible operations such as I/O. Numerous alternatives have been studied [29, 31], but there is currently no satisfactory solution, especially for server applications.
- There is no commodity hardware support for transactional memory and software implementations have significant performance overheads even when transactions are contention-free [37].

Transactional memory is an extremely active area of research and a number of these problems may well be resolved in the future. McKenney *et al.* [21] contrast transactional memory with locks and provide a more detailed view of their respective advantages and disadvantages.

```
cilk int fib (int n)

    if (n < 2) return n;
    else

        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
```

Figure 2.1: The Cilk implementation of the Fibonacci program where the recursive calls are executed in parallel.

2.1.3 Futures

A future, or promise, is a construct that acts as a proxy for a result that is not known, because it has not yet been computed. Futures may be implicit, in which case they are resolved at the point of use, or explicit, which requires the programmer to resolve them before use. There are other possible variations, so a specific implementation is best used for exposition.

Figure 2.1 shows a Cilk [2] program to recursively compute Fibonacci numbers in parallel.

The `cilk` keyword identifies a procedure that may spawn sub-procedures in parallel and synchronize upon their completion. The `spawn` keyword indicates that the following function call may be executed in parallel with the caller; this also makes the value returned by the call a future. Futures are resolved explicitly with the keyword `sync`.

With Cilk, a programmer expresses potential concurrency and leaves it to the system to determine how many threads to use, how to balance load, and how threads communicate. The Cilk abstractions ease parallel programming, but they do not aid a programmer in the difficult and error-prone activity of identifying shared resources and synchronizing access to them. Cilk only offers locks, with all their attendant issues.

There is much ongoing research into futures. Navabi *et al.* [23] use program analysis to safely execute Java applications that are annotated to use futures. Chan and Abdelrahman [3] create a thread for every method invocation in a Java program. A combination of compile-time and runtime analyses detect and enforce dependencies and preserve sequential semantics. Swaine *et al.* [34] offer a futures-based approach to incrementally parallelize the runtime systems of high level languages.

2.1.4 Event-driven Programming

In an event-driven program such as `thttpd` [35], a main loop receives *events* from the operating system and invokes *handlers* that have been registered to process these events. A server registers a handler for an event, *e.g.* an incoming client connection. A handler typically performs a short, non-blocking task, then invokes an asynchronous service and registers another handler to be called when that service completes. In this manner, a client request is processed by a series of handlers.

A simplified example is shown in Figure 2.2.

Event-driven servers provide I/O concurrency, as multiple requests may be in flight at once. In Figure 2.2, the `Accept_Connection` handler would be invoked by the event loop for a new connection. This handler accepts the connection and registers the `Read_Request` handler for invocation when data is received on the new connection. However, the event loop might invoke the `Accept_Connection` handler again on another incoming connection before calling the `Read_Request` handler for the first connection.

Event-driven servers typically do not provide CPU concurrency as the event loop is serial. It is implicit in this model that handlers run atomically with respect to one another: a handler is guaranteed to run to completion without interference from other handlers. While this makes it easier for developers to reason about program logic, these event-driven servers cannot exploit multi-processors. Furthermore, an event handler must not block, as this would cause the entire server to block.

An extension to the event-driven model to exploit multi-processors has been described [39] in which programmers manually specify *colors* for handlers to explicitly enable parallelism. Handlers of a given color run atomically only with respect to other handlers of the same color. This concurrent execution improves performance, but if the programmer specifies colors incorrectly, unsafe parallelism may occur. Further, a simple color is insufficiently expressive: if handler X conflicts with handler Y, and handler Y conflicts with handler Z, all three handlers must be given the same color, even though handler X may not conflict with handler Z.

Gaud *et al.* [14] improve the event coloring system's workstealing algorithm, but do not address the limitations of the model.

SEDA [36] runs event handlers concurrently, but offers no direct support for programmers to avoid concurrency related errors.

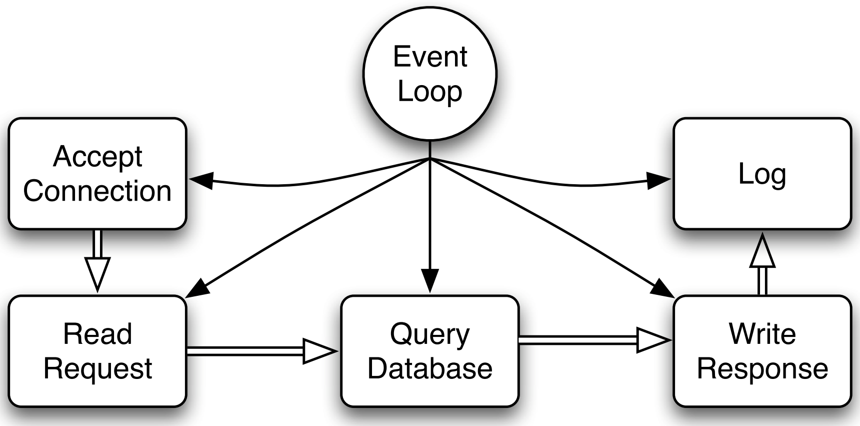
2.1.5 Message passing

The parallel programming models discussed thus far address threads that share state. When threads do not share state, they must communicate by passing messages. This type of interaction may be modeled by process calculi such as CSP [17], the study of which constitutes an area of research in itself.

This dissertation is concerned with shared state concurrency and does not consider message passing models further as their issues are quite different.

2.2 Other Related Work

The well known problems of parallel programming and its growing importance have spurred a great deal of research in this area. Much of this research focuses on improvements to the parallel programming models discussed earlier in this chapter, and is described there. This section addresses other closely related work.



```
main()
  fd = socket(...);
  ...
  listen(fd, ...);
  event_reg(..., Accept_Connection, fd);
  event_loop();

Accept_Connection(..., fd)
  ...
  sess = alloc_session(...);
  sess->conn_fd = accept(fd, ...);
  event_reg(..., Read_Request, sess);

Read_Request(..., sess)
  ...
  read(sess->conn_fd, sess->buf, ...);
  event_reg(..., Query_Database, sess);

...
```

Figure 2.2: A simplified event-driven server processes client requests with a chain of handlers, each registered by the preceding handler to operate on the request in turn. The session holds state, and is passed along the chain, with each handler updating it as needed.

Autolocker [20] uses program analysis to correctly convert atomic sections into lock based code, given annotations to associate locks with variables and identify atomic sections. Cherem *et al.* [4] only require that atomic sections be specified, and add multi-granularity locks. Emmi *et al.* [9] focus on optimizing the set of locks acquired for annotated critical sections.

Flanagan and Qadeer [13] have developed a static analysis to verify atomicity compliance in annotated Java programs. Flanagan and Freund [11] extended this work to detect and insert missing synchronization operations in a similarly annotated Java program.

Our work differs from all these in specifically avoiding the need for any annotations. We focus on servers, a class of applications that has innate, obvious parallelism, written to a programming model with implicit atomic blocks.

Chapter 3

Safe Parallelism

A core tenet of this work is that parallelism should be safe by default: unsafe parallelism must be prohibited. In such a system, programmer error will prevent potential concurrency which is a performance problem, not a correctness problem.

3.1 Approach

This system builds on the event-driven model. A key advantage of this model is that event handlers run atomically with respect to each other. This allows the programmer to design her application with serial execution in mind, avoiding the need for error-prone reasoning about concurrency.

Our approach combines a static analysis that generates constraints on the concurrent execution of event handlers with a runtime system that enforces these constraints to enable safe concurrent execution.

The system therefore has two components:

1. A static analysis runs on the event-driven server application's source code and conservatively determines if event handlers conflict with each other. One or more conflicts between a pair of event handlers creates a constraint on the concurrent execution of those handlers. The static analysis produces a set of these safety constraints.

2. A runtime scheduler uses these constraints to safely run event handlers concurrently. It simultaneously gathers statistics on wait and run frequencies and durations for each event handler.

An initial run on an unmodified event-driven server may provide marginal performance gains. Many such servers use global variables extensively; such accesses cause conflicts that result in constraints on concurrent execution. The programmer will need to modify the server to remove conflicts between event handlers, thereby eliminating constraints on their concurrent execution and obtaining larger performance gains.

To this end, both components of the system provide detailed information to the programmer. The static analysis specifies the type and location of conflicts between event handlers, while the runtime scheduler provides information on which constraints actually cause the most slowdown at runtime. The programmer can focus on removing those conflicts that cause the most damaging constraints.

3.2 Safety Constraints

This section discusses the constraints that enable safe parallelism, and how they are found and enforced.

3.2.1 Static Analysis

The analysis must determine when data is shared between handlers in ways that preclude safe parallelism, *i.e.* when the set of data written by one handler overlaps with the set of data read or written by another handler. This analysis is conservative—a handler that *might* access a data object must be treated as though it will.

Independence: Different event handlers are invoked at different times for different requests. Each handler runs to completion on its invocation, thus the per-request state cannot be stored on the stack. Instead, this state is usually stored in a data structure that is specified by the application when it registers a handler. This "session" data structure[1] is then passed to the handler by the event management system when the handler is invoked. The application may use just one type of session or different session types for every event handler.

Thus, the invocation of an event handler, $f(s)$, has two *roots* from which all potentially unsafe accesses may take place: the global variables of the program, and the session, s. From these roots arise two sets of reachable objects, G and S. For precision, reads and writes must be distinguished, thus the sets of globals and session elements actually read and written by event handler f are G_f^R, G_f^W, S_f^R and S_f^W respectively.

Let the total set of data objects read and written by event handler f be T_f^R and T_f^W, where $T_f^R = G_f^R \cup S_f^R$, and $T_f^W = G_f^W \cup S_f^W$.

We are interested in the conflicts between event handlers f and g, which is $X_{f,g}$, the set of data objects that are accessed *unsafely.* This excludes those data objects that are only read by both handlers, but includes those that are read by one and written by the other, as well as those written by both.

$$X_{f,g} = (T_f^W \cap T_g^W) \cup ((T_f^R \cap T_g^W) \cup (T_g^R \cap T_f^W)) \tag{3.1}$$

[1]This data structure is often called the "context", a term that has another meaning in program analysis; we are using "session" instead to avoid confusion.

The analysis must compute $X_{f,g}$ for all pairs of event handlers f and g. There are three possible outcomes:

1. $X_{f,g} = \emptyset$, which implies that there are no conflicts. In this case, the handlers f and g may be scheduled concurrently without conflict; there is no safety constraint.

2. $\exists v \in X_{f,g} : v \in G$, which implies that at least one of the data objects accessed unsafely is a global variable. The handlers f and g may *not* be executed concurrently under any circumstances; the safety constraint is absolute.

3. $\exists v \in X_{f,g} : v \in S$ and $\forall v \in X_{f,g} : v \notin G$, which implies that there is a conflict on a session element, but not on a global variable. This case is more subtle as the safety constraint is conditional.

Session conflict: Since this analysis runs statically, it cannot take into account the specific sessions passed to a handler. But when two handlers conflict only on session elements, they could be executed concurrently, *if* it were known at invocation time that the specific sessions that would be passed to them are different.

Sessions in event-driven server applications usually represent client requests. As such, they tend to be independent of each other for the most part. The ability of the analysis to distinguish between conflicts on globals and conflicts on session elements is important in uncovering parallelism as, absent this distinction, session conflicts would prohibit concurrency between event handlers even when invoked with different sessions.

Output: The safety constraints discovered by the analysis must be made available to the runtime system. Thus, the analysis must output the constraints in a form that is suitable for compiling into the application.

Additionally, the analysis must report its results to the programmer in a form that allows her to perform incremental improvement, *i.e.* by removing the conflicts that create a safety constraint.

For instance, a conflict may be caused by a rarely taken error handling path that increments a statistics counter. One solution to this would be to use an atomic operation to perform the increment; this will be recognized by the analyzer as atomic and therefore thread-safe.

Another general solution is to convert global data into private data whenever possible, using a single handler to manage all accesses to that data.

3.2.2 Runtime Scheduling

The runtime system is responsible for executing the analyzed server application with maximal concurrency, subject to the constraints identified by the analysis.

If the analysis specifies that two handlers never conflict, the scheduler is free to schedule those handlers simultaneously. If the constraint states that the handlers always conflict, the runtime system must never execute both handlers simultaneously. And finally, if the constraint states that

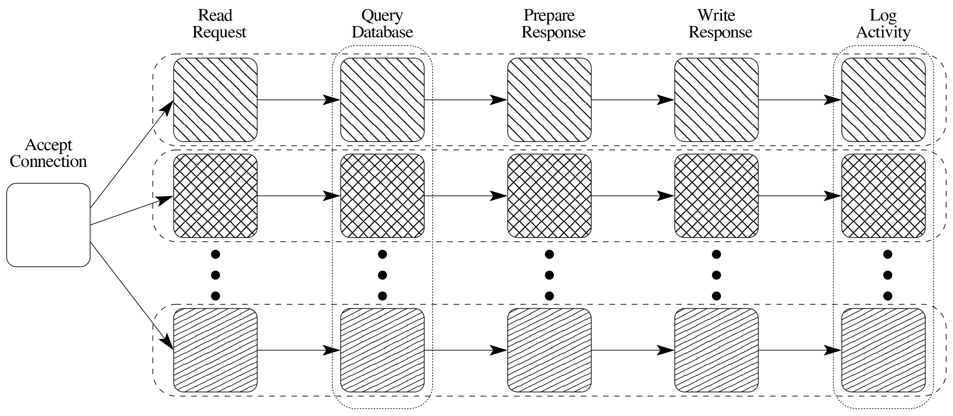


Figure 3.1: A simplified event-driven server processes client requests with a chain of six handlers.

the handlers conflict on the session, the runtime system is free to execute them simultaneously if and only if the invocations have different sessions. To achieve this, the scheduler must dynamically determine whether a session is in use by a currently executing event handler.

Statistics: The programmer improves the performance of the server application by increasing available safe parallelism. This is accomplished by removing constraints on the concurrent execution of event handlers by eliminating conflicts reported by the static analysis.

However, not all constraints are equal. For instance, a maintenance event handler may conflict with every other event handler, but run only once every 5 minutes. The overall performance gain from removing conflicts with such an event handler is likely to be so small that the programmer may choose to ignore this constraint entirely.

Other constraints may not be as obvious in their effect on performance. Thus the scheduler must provide statistics on the execution of a server to enable the programmer to determine serial bottlenecks and prioritize constraint removal.

3.3 Example Server

Figure 3.1 shows another view of the event-driven server described in Figure 2.2. The Query handler conflicts with itself due to the use of a global, as does the Log handler. These conflicts are depicted in the figure by the vertical grouping of these handlers. All the other handlers conflict with each other on session elements.

The set of constraints resulting from these conflicts are summarized in Table 3.1.

	Read	Query	Prep	Write	Log
Read	s	s	s	s	s
Query		g	s	s	s
Prep			s	s	s
Write				s	s
Log					g

Table 3.1: A table showing constraints for the event-driven server in Figure 3.1. *g* indicates conflicts on globals. The Query handler updates a global and thus conflicts with itself. Global conflicts supersede conflicts on session elements indicated by *s*. The Read handler accesses a session element that is updated by the Accept handler and thus the two handlers have a session conflict.

Chapter 4

elyze

`elyze` is a static analysis tool built on the CIL analysis framework [24]. It runs on event-driven server code written in C to the `libevent` interface [28] and finds constraints on the concurrent execution of the server's event handlers.

4.1 Approach

The programming languages most often used when developing server applications are C, C++ and Java. We chose to begin with C, as many of the most popular server applications are written in it.

A key consideration in developing any static analysis is the approach to discovering aliasing. There are a number of choices of algorithm, each varying in precision and speed [32, 1, 5]. We were unable to locate an implementation of an alias analysis that was sufficiently well documented so as to be extensible, and also could satisfy our specific requirements:

- Fast enough to process large code bases ($\tilde{1}$00,000 lines) in reasonable time.
- Precise enough to avoid unnecessarily conservative assumptions.
- Able to distinguish accesses to different structure fields.

We were therefore compelled to implement an alias analysis. We did so atop the CIL analysis framework [24], which enables the traversal of an application's complete abstract syntax tree, including all globals, functions, statements, expressions, etc, and further allows the generation of definitions that will be compiled into the application.

4.2 Overview

`elyze` begins by identifying the event handlers in the program together with the sessions to be used for their invocation; it does so by looking for `libevent` handler registration calls as shown below.

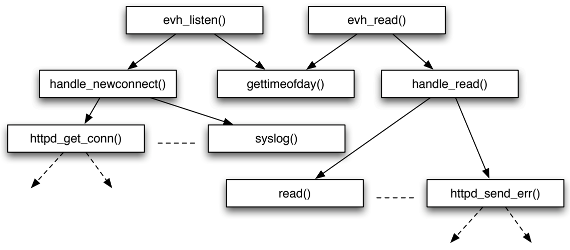
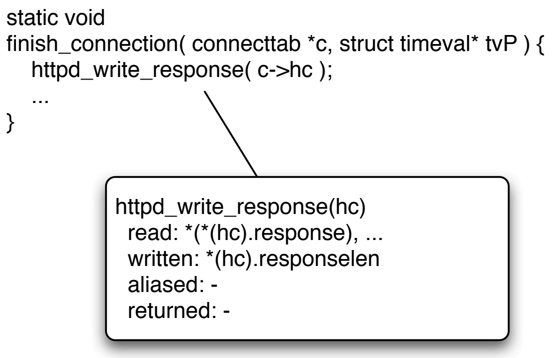


Figure 4.1: `elyze` builds the call graph for each event handler.



```
static void
finish_connection( connecttab *c, struct timeval* tvP ) {
   httpd_write_response( c->hc );
   ...
}
```

```
httpd_write_response(hc)
  read: *(*(hc).response), ...
  written: *(hc).responselen
  aliased: -
  returned: -
```

Figure 4.2: `elyze` applies function summaries at call sites.

```
main() {
  ...
  event_set( &ev_listen4, hs->listen4_fd,
     EV_READ|EV_PERSIST, evh_listen, (void*) 0 );
  ...
}
handle_newconnect( struct timeval *tvP, int listen_fd ) {
  ...
  event_set( c->ev_fd, c->hc->conn_fd,
     EV_READ, evh_read, (void*) c );
  ...
}
```

For each event handler, a call graph is generated as shown in Figure 4.1.

Next, `elyze` performs a context-sensitive, summary-based analysis on each call graph that begins at the bottom of the graph and works upwards to the event handler. For each function, the analyzer generates a summary identifying the data objects read, written, and aliased, as well as the data objects returned, if any. This summary is applied at each call site, as shown in Figure 4.2.

On reaching the top of the call graphs, `elyze` has identified the sets of data objects that are read and written by each event handler. `elyze` then evaluates Equation 3.1 to determine conflicts between each pair of event handlers, and outputs its results which are detailed in Section 4.2.8.

4.2.1 Context sensitivity

`elyze` follows Steensgard's algorithm [32] in treating assignments bi-directionally using unification. This approach trades trades precision for speed, in contrast to Andersen's algorithm [1], which handles assignments using subtyping and gains precision at the cost of speed.

Das has shown [5] that handling the most common case of pointer use in C programs, namely passing the address of an updateable value as an argument to a procedure, leads to a significant increase in precision of Steensgard's algorithm, almost to the level of Andersen's algorithm. Das accomplishes this improvement by using subtyping at call sites when assigning the arguments provided by the caller to the callee's parameters.

`elyze` achieves the same benefits in precision by being context sensitive, *i.e.* distinguishing between different calls to the same function. We use a bottom-up, summary-based approach to context sensitivity, similar to Nystrom *et al.* [7]. Thus, our analyzer produces a function summary composed of four lists containing those data objects that are read, written, aliased, and returned. As the analysis moves up the call graph, it *applies* the summary of a function at each call site in a calling function. Application refers primarily to three operations:

1. Adding all globals from the callee's summary into the caller's summary.
2. Adding all arguments in the callee's summary, after appropriately translating their names, into the caller's summary and creating any required alias relationships.
3. Creating all required alias relationships caused by assignment to the callee's return value.

In contrast to other approaches to context sensitivity such as function inlining or argument replication, this method is efficient in both time and space.

4.2.2 Structure fields

It is important for `elyze` to distinguish accesses to the different fields of a structure, particularly the session data structure which tends to be a collection of variables that are stored in an aggregate for convenience. It is often the case that different event handlers access different parts of the session.

```
static void handle_read(connecttab *c, ...)
  ...
  c->active_at = tvP->tv_sec;
  ...

static void handle_send(connecttab *c, ...)
  ...
  c->active_at = tvP->tv_sec;
  ...

static void handle_linger(connecttab *c, ...)
  ...
```

```
r = read(c->hc->conn_fd, ...);
...
```

The code snippets above are from `thttpd`. Three event handlers are shown; the `connecttab` pointer is the session. `handle_read()` and `handle_send()` both update the activity statistic counter in the session; this causes `elyze` to report a conflict on that session element for these two handlers. However, `handle_linger` does not touch the activity statistic counter and does not conflict with the other two handlers on this data object.

Many previous alias analyses do not distinguish such accesses; this is considered a difficult problem for C because of type-casting, the ability to take the address of a field, and the widespread use of a static heap model [26]. Without the ability to distinguish these accesses, all three event handlers would be deemed to conflict with each other, unnecessarily restricting potential concurrency.

`elyze` models structure fields as normal variables, while an aggregate variable itself is modeled both as a normal variable and as a container of other normal variables. We do not attempt to handle casts from one aggregate type to another but simply act conservatively (assuming all-to-all aliasing).

We deal with recursive types by *k-limiting* [19]: we artificially limit the depth to which we parse aggregates (we chose $k = 2$ based on experiments). An aggregate variable at a depth of 2 is treated as a normal variable which additionally represents all its fields without regard to their concrete type.

We have experimentally validated this approach to field sensitivity and find that it produces sufficient precision for our purposes (*i.e.* handlers are not constrained from concurrent execution unnecessarily), without the performance impact of more sophisticated methods [6].

4.2.3 Arrays and buffers

In contrast to the treatment of structure fields, `elyze` treats both buffers and arrays as scalars. Even with the inclusion of value flow analysis, it is difficult to statically distinguish accesses to different parts of the same buffer, or to different indices of an array, and thus `elyze` makes no attempt to do so.

4.2.4 Recursion

A recursive call creates a cycle in the call graph. Thus, a bottom-up analysis such as `elyze` will encounter a call to a function it hasn't seen before (for which no summary is present). `elyze` handles recursion by iterating over recursive call chains to a fixed point [8].

Extensive use of recursion in a large application can cause many iterations over the call graph, which can result in long analysis times. Caching and re-validating analysis results across runs will help address this issue.

4.2.5 Function pointers

Alias analysis must be completed in order for an indirect call through a function pointer to be resolved. Thus, in order to completely build the program call graph, alias analysis must be completed. However, performing alias analysis requires traversal of the program call graph.

`elyze` handles this cyclic dependency with the well known iterative approach [8, 7], *i.e.* it proceeds with the alias analysis, identifying the functions that a function pointer may resolve to and updating the call graph as needed. On completing analysis of the initial call graph, `elyze` processes any expansions of the call graph before re-analyzing those functions from which indirect calls are made. These steps are repeated until the results stabilize.

4.2.6 Explicit summaries

For soundness, an alias analysis usually must operate on an entire program, otherwise the effects of external functions are not seen. Instead, our summary-based approach allows for the presence of external functions, provided a summary for each such function is already available. Since this summary is simply the set of externally visible effects of the function, it is fast and easy to write by hand. We have written 125 summaries for various `libc` functions, and they are three lines each on average.

There are many advantages to this facility. Reducing the total code that must be analyzed naturally speeds up the analysis. Another advantage is that summaries can be created even of functions for which source is unavailable, such as system calls. And further, there are complex functions (such as `malloc()`) that are hard to analyze and would require unnecessarily conservative assumptions for safety.

Standard library implementations of `malloc()` and `free()` manage a number of large buffers. `malloc()` returns a pointer index into one of these buffers. Even with value flow information, it would likely be impossible to statically distinguish the different offsets into the various memory pools. A static analysis would conservatively determine that all calls to `malloc()` return an alias of a global and that all the returned pointers are therefore aliases of each other.

If we recognize that from a caller's standpoint, it is only necessary to know that the implementation is reentrant and the return value is a "fresh" pointer, we can insert a summary for `malloc()` and avoid the problem.

4.2.7 Variables of interest

`elyze` considers only those statements and expressions that contain a variable of interest. For a given function, when analysis begins, the only variables of interest are the globals of the program, and those parameters that may be side-effecting, *i.e.* those that are or contain pointers. A local variable becomes interesting when it is aliased with a variable of interest.

```
static int check_referer(httpd_conn *hc)
  char *cp = "...";
  char **cpp = &cp;
  ...
  cp = hc->hostname;
  ...
```

The code snippet above is from `thttpd`. On entry into the function, the analyzer will only consider `hc` (and its elements) interesting (besides the globals, which are always interesting). The local `cp` is ignored until it is aliased with an element of the argument, at which point it is marked interesting. This will cause the function to be re-analyzed, to look for any aliasing of `cp` prior to this assignment (the use of `&cp`).

This approach significantly limits the number of data objects that the analyzer needs to track, and speeds up the analysis. However, when an aliasing assignment is made between a variable of interest and a local variable, that local variable must be marked interesting and the function re-analyzed to explore any aliasing activity involving the local prior to this assignment. We complete the analysis of the function before redoing it to gather multiple interesting locals in one pass. A chain of assignments from local to local, and eventually to an interesting variable can cause the function to be analyzed repeatedly (as many times as there are assignment statements in the function, in the worst case). In practice, we find that in `thttpd`, 35% of the functions are analyzed twice, and 1.5% are analyzed three times; yielding an average of 1.38 iterations per function. Nonetheless, the net effect of avoiding unnecessary analysis of uninteresting variables is a performance benefit.

4.2.8 Output

When `elyze` completes analysis of an event-driven application, it outputs its results in two forms: a compilable format for use by `multivent`, and human readable conflict information for use by developers seeking to increase concurrency.

For `multivent`

For each event handler, `elyze` generates two access tables. The first table identifies global conflicts against every other handler, distinguishing between read and write conflicts. The second table identifies the session elements accessed by the handler, again distinguishing between reads and writes.

Chapter 5 describes these tables further.

For the programmer

The programmer must be guided in increasing parallelism, *i.e.* making changes to allow event handlers to run in parallel. To aid in this, `elyze` outputs constraint information on each event handler pair. The following is an extract from `elyze`'s output for Tor showing the three possibilities:

```
signal_cb() and signal_cb() always conflict: ...
signal_cb() and rotate_cb() always conflict: ...
...
rotate_cb() and evdns_request_timeout_cb() do not conflict.
...
nameserver_prod_cb() and cpuworker_cb() conflict on context: ...
...
```

For each constraint, `elyze` lists the variables on which conflicts exist (variable names not shown above due to limited space).

The programmer must first determine which constraints are most damaging to performance; she is aided in doing so by `multivent`'s dynamic analyses (described in Section 5). After identifying a constraint that she would like to remove, *i.e.* selecting a pair of event handlers that she would like to run concurrently, the programmer considers each of the conflicts that cause the constraint. She must identify every conflicting variable access and determine how to remove the conflict. `elyze` assists with locating accesses by displaying the summary of every function in the application, as shown below:

```
signal_cb
---
called from:
 control_signal_act()

read:
 ...
 g:*(current_consensus) at routerlist.c:1054
  via alias *(consensus).routerstatus_list created at ...
  via alias *(tmp___0) created at routerlist.c:1038
  through call to router_pick_directory_server_impl() at ...
  through call to router_pick_directory_server() at ...
  through call to directory_get_from_dirserver() at ...
  through call to authority_certs_fetch_missing() at ...
  through call to update_certificate_downloads() at ...
  through call to update_networkstatus_downloads() at main.c:1337
  through call to do_hup() at main.c:1581
  ...

written:
 ...
 g:policy_root.hth_n_entries at policies.c:527
  via alias *(head).hth_n_entries created at policies.c:1288
  through call to addr_policy_free() at policies.c:1275
  through call to addr_policy_list_free() at routerlist.c:2349
  through call to routerinfo_free() at routerlist.c:2678
  through call to routerlist_remove() at dirserv.c:800
  through call to directory_remove_invalid() at dirserv.c:278
  through call to dirserv_load_fingerprint_file() at main.c:1322
  through call to do_hup() at main.c:1581
  ...
```

```
aliased:
  ...

fps_called:

returns:

dangerous:
```

With this information, the programmer can identify all conflicts that prevent a pair of event handlers from being executed concurrently, and also locate each conflicting access. She can then determine how to eliminate the conflict. This process is detailed for both `thttpd` and `tor` in Chapter 6.

4.3 Limitations

There are two possible sources of unsoundness in `elyze`:

1. C supports the use of inline assembly, which may be used to unsafely share data between handlers. We are not aware of any solution to this problem proposed in the literature. Currently, we issue a warning when we observe any inline assembly.

2. Resources other than memory can cause conflicts: if two handlers write to the same file descriptor, that is a conflict that `elyze` will not recognize. The conservative solution to this problem is to prohibit concurrent execution of those handlers that issue system calls using file descriptors. This is not a viable option, as event handlers in server applications tend to work extensively with file descriptors.

   This can be addressed by treating file descriptors as special pointers which can be aliased and are indirected via system calls. Thus, a handler using the `write()` system call on a file descriptor will be writing to the abstract object "pointed at" by the file descriptor, and will conflict with another handler using the same file descriptor.

Chapter 5

multivent

`multivent` is a multi-threaded event management library that provides the runtime scheduling features required for safe parallelism. Designed to complement **`elyze`**, **`multivent`** uses multiple threads to run event handlers concurrently, subject to **`elyze`**'s safety constraints.

5.1 Approach

There are a number of possible approaches to the design of the runtime scheduler. A naïve design would create a number of threads to invoke handlers with each thread selecting a non-conflicting handler to invoke. This would exhibit very poor performance, as each thread would need to query and update a shared scoreboard for the handlers being executed.

Another approach is the color scheduler of Zeldovich *et al.* which avoids the shared scoreboard by using separate queues for each non-conflicting handler. However, this scheduler is unable to express the constraints shown in Table 3.1 for the example server described in Section 3.3, where the Log handler may not be executed concurrently with another handler for the same session, *or* with another Log handler.

This limitation could be addressed by introducing a second level, a *flavor*. A color/flavor scheduler would ensure that no two handlers of the same color or of the same flavor execute concurrently.

A more flexible alternative was chosen: a lock based approach, in which a readers-writer lock is associated with each conflict. An invocation thread about to execute a handler acquires all the locks required to protect the objects accessed by that handler (in a canonical order, to avoid deadlocks) before invoking the handler. This approach is detailed further in the rest of this chapter.

5.2 Overview

The manual page for the `libevent` library [28] describes the event API as providing a mechanism to execute a callback function when a specific event on a file descriptor occurs, or after a given time has passed.

Like other event management libraries, `libevent` abstracts away the specific OS mechanism for asynchronous event notification (`select()`, `poll()`, `kqueue()`, etc.) and provides additional infrastructure for receiving signal and timeout notifications.

An application initializes the library (`event_init()`), sets up one or more events (`event_set()`), and registers them (`event_add()`) before entering the library's event loop (`event_dispatch()`) from which callback functions are invoked.

`multivent` additionally starts multiple threads to invoke event handlers with a job queue for each. It also creates a readers-writer lock for each potential conflict, and gathers statistics on event handler execution.

5.2.1 Thread Behavior

When a server application calls `event_dispatch()` to enter the event loop, `libevent` uses that thread to check for timeout expiry, for asynchronous events, and to invoke handlers that are ready to execute.

`multivent` uses the dispatch thread to test for ready handlers, but does not invoke them directly. Instead, it chooses an invocation thread by simple round robin and delivers the event information to the chosen thread's job queue.

An invocation thread receives an event in its queue, determines the event handler that must be invoked, and acquires the necessary locks before invoking the handler.

5.2.2 Satisfying Safety Constraints

In order to satisfy the safety constraints specified by `elyze`, the library uses readers-writer locks for each potential conflict.

In order to handle absolute safety constraints, *i.e.* when two handlers must never be executed concurrently, a readers-writer lock is created for each pair of handlers. When a handler is to be invoked, the responsible thread acquires the lock it shares with each other handler for which an absolute safety constraint is specified. `elyze` specifies for each such constraint whether the handler is a reader or a writer, which allows the correct type of lock acquisition to be made.

Session safety constraints are satisfied similarly. A server specifies the session to be used when setting up an event. For each unique session seen by `multivent`, a number of readers-writer locks are created; this number is specified by `elyze` and is the number of fields parsed from the largest of the types used for sessions by the server.

This approach removes the need for session identity checks—if two handlers that conflict only on session elements are invoked concurrently with the same session, both invocation threads will attempt to acquire the same locks and only one will proceed. If the handlers are invoked with different sessions, they will run concurrently as the threads will acquire different locks.

5.2.3 Thread Count

The number of threads that `multivent` uses for event handler invocation depends on a few criteria:

- The number of cores in the target machine limits the amount of true parallelism available.
- The use of blocking calls in the application's event handlers ties up threads.
- The size and number of the application's event handlers also impact the desired number of threads. Many short event handlers could use more threads than a few long event handlers.

Thus, `multivent` allows the number of threads that are started to be configured by an environment variable. The default value is 8, thus on a quad-core machine, `multivent` starts 32 threads. Our experience suggests that this is a reasonable default, but a reasonable extension could monitor progress and dynamically optimize the size of the thread pool.

5.2.4 Dynamic Analysis

`multivent` performs some dynamic analysis of the server applications that it executes. The library gathers a number of statistics that are intended to aid the programmer as she works to improve server performance.

For each event handler, `multivent` records:

- Number of times invoked.
- Minimum, maximum, and mean time for completion.
- Minimum, maximum and mean wait time, *i.e.* from event firing until invocation.
- Maximum number of pending invocations.

In addition, `multivent` records when contention occurs, the data object on which it occurs, and the minimum, maximum, and mean duration.

Gathering these statistics, especially those related to contention, adds some overhead. Thus, `multivent` checks environment variables to determine whether statistics collection is to be enabled.

From these statistics, the programmer can prioritize conflict removal. An event handler that is invoked frequently and has a high mean time for completion has a higher cost for conflicts that an event handler that runs occasionally, or completes quickly.

While this data is valuable, understanding the application's design is also important as can be seen in the evaluation of the system.

Chapter 6

Evaluation

6.1 Evaluation Criteria

A meaningful evaluation of `elyze` and `multivent` must consider a number of different aspects:

- Does the system work with real-world server applications?
- Does analysis complete in reasonable time?
- Does the system aid the programmer in identifying and locating conflicts which must be removed to improve performance?
- Does the system eliminate parallelism related bugs?
- Does use of the system allow improved server performance?

The remainder of this chapter answers these questions by describing the use of `elyze` and `multivent` on two real-world server applications: the web server `thttpd`, and the onion router `tor`.

All tests were carried out on quad-core machines: Athlon Phenom II X4 955 processors at 3.2 GHz with 4 GB RAM.

6.2 Test Cases

In addition to testing `elyze` on real servers' code, we have developed a test framework composed of a simple event-driven server application, within which we have created about 45 test cases that test `elyze`'s ability to recognize some more unusual code constructs such as:

- functions creating aliases between their arguments,
- passing aliases through typecasts,

- functions returning `struct` variables by value,
- passing aliases through unions,
- mutual recursion, and
- compound global initializations.

6.3 Safe Parallelism for `thttpd`

`thttpd` (tiny/turbo/throttling HTTP server) is described by the author as a simple, small, fast and secure HTTP/1.1 server. It is a single-threaded, event-driven server application that uses its own event management framework. The server source code consists of 7496 lines of C code[1].

6.3.1 Motivation

`thttpd` does not perform any CPU-intensive activity and thus may not seem like a suitable server application to parallelize. However, current UNIX kernels do not offer any standard asynchronous disk I/O interface, and so `thttpd`'s single thread must block on synchronous disk I/O calls.

When the server receives a request for a file, it uses `mmap()` to map the file into memory and `write()` to send it out over the network. `write()` will trigger page faults when a block has been mapped but not paged in, and these faults block the server's single thread.

Thus `thttpd` is ideally suited to serve static data from its memory cache; for this type of workload, this is one of the fastest web servers available. When the data to be served cannot be entirely cached in memory, `thttpd`'s performance degrades as the size of the files it must serve increases.

If multiple threads could concurrently execute the event handler that performs the blocking `write()` call, then server freezes could be avoided, together with their impact on performance.

The goal then, was to improve `thttpd`'s performance on workloads involving large files that are not in cache. This was to be accomplished by using `elyze` and `multivent` to parallelize `thttpd`, specifically ensuring that the event handler that triggers page faults could have multiple threads executing it simultaneously.

6.3.2 `elyzing`

Before analysis: `thttpd` uses its own event management code: a module named `fdwatch`. It also uses its own timer management code and installs signal handlers using the OS interface.

The first task was to replace these modules with `libevent`, which proved to be a straightforward mechanical exercise. There was no performance impact from this change.

[1]Estimated using David A. Wheeler's 'SLOCCount'.

Handler	Runs	Mean	Minimum	Maximum
listen	29	3061.45	0s 25us	0s 79683us
tidle	95	36.0842	0s 18us	0s 82us
toccasional	7	14296.3	0s 62us	0s 43423us
sigalrm	2	6	0s 5us	0s 7us
clear_conn	650	38.4877	0s 10us	0s 535us
read	650	4188.12	0s 24us	0s 98246us
send	21273	172969	0s 9us	6s 210315us

Table 6.1: `multivent` dynamic analysis results for `thttpd` show the number of invocations and the mean, minimum and maximum times for completion of each event handler.

Analyzing: `elyze` completely analyzes `thttpd` in $\tilde{2}$.4 seconds. It then informs us that:

```
...
evh_send() and evh_send() conflict on context: *(c).conn_state ...
...
```

The function `evh_send()` performs the blocking `write()` call with which we are concerned. Out of the box, this event handler only conflicts with itself if the session (context) is the same; this constraint allows `multivent` to run multiple `evh_send()`s in parallel so long as the session passed in to each invocation is different.

As this was precisely the goal, we ran performance tests on this version of `thttpd` against the original. For large workloads, we found an approximate 12% speedup. Details of these tests are in Section 6.3.3. It must be emphasized that this speedup was achieved without any further code changes: simply running the server through `elyze` and `multivent` produced this increase in performance.

Going further: `multivent`'s dynamic analysis (shown in Table 6.1) informed us that our new `thttpd` could benefit from the parallelization of the `evh_read()` event handler (handler names are truncated in the following table):

Previously, `elyze` had told us that:

```
...
evh_read() and evh_read() always conflict: ...
...
```

For the next step, we chose to work on removing the constraint preventing multiple copies of the `evh_read()` event handler from running concurrently. `elyze` reported that the constraint preventing this was caused by 123 conflicts. Examining the summary of this event handler showed that these conflicts fell into four categories:

1. Writes to static variables in various functions in `libhttpd.c`, mostly used to record `malloc()`ed buffer pointers and their sizes. Each of these was removed, either by statically allocating the buffers, or by using atomic operations.

2. Functions to look up the weekday and month in `tdate_parse.c` were performing a one-time sort of their tables on the first invocation. This was removed after pre-sorting the tables.

3. An internal version of `inet_ntoa()` that used an internal static buffer was adjusted to take a buffer as an argument.

4. All the remaining conflicts derived from a call to `mmc_map()`, at the end of `evh_read()`. This was resolved by splitting off this call into its own event handler, `evh_map()`, and triggering that event handler from `evh_read()`.

These changes allowed multiple copies of `evh_read()` to run in parallel, improving the modified `thttpd`'s performance on workloads with large files to 15%. Details on these results follow.

6.3.3 Results

We used `http_load` [18] to simulate 200 clients fetching a total of 4GB of data. Each 4GB workload was composed of files of the same size, ranging from 512KB to 6MB. We averaged the results of multiple tests on each of the following versions of the server:

1. Unmodified `thttpd`

2. `thttpd` with `multivent`

3. `thttpd` with parallelized `evh_read()`

The results of these tests, in Figure 6.1, show that as file size increases, the parallelized `thttpd` performs better.

6.4 Safe Parallelism for `tor`

`tor` is a system intended to enable online anonymity. It is an implementation of onion routing which works by relaying communications over a network of servers run by volunteers in various locations. `tor` is implemented as an event-driven server using `libevent` in 74,600 lines of C code[2].

Users of a `tor` network run an onion proxy on their machine. The `tor` software periodically negotiates a virtual circuit through the `tor` network, using multi-layer encryption, ensuring perfect forward secrecy. At the same time, the onion proxy software presents a SOCKS interface to its clients. SOCKS-aware applications may be pointed at `tor`, which then multiplexes the traffic through a `tor` virtual circuit. Once inside the `tor` network, the traffic is sent from router to router, ultimately reaching an exit node at which point the cleartext packet is available and is forwarded on to its original destination. Viewed from the destination, the traffic appears to originate at the `tor` exit node.

[2]Estimated by David A. Wheeler's 'SLOCCount'.

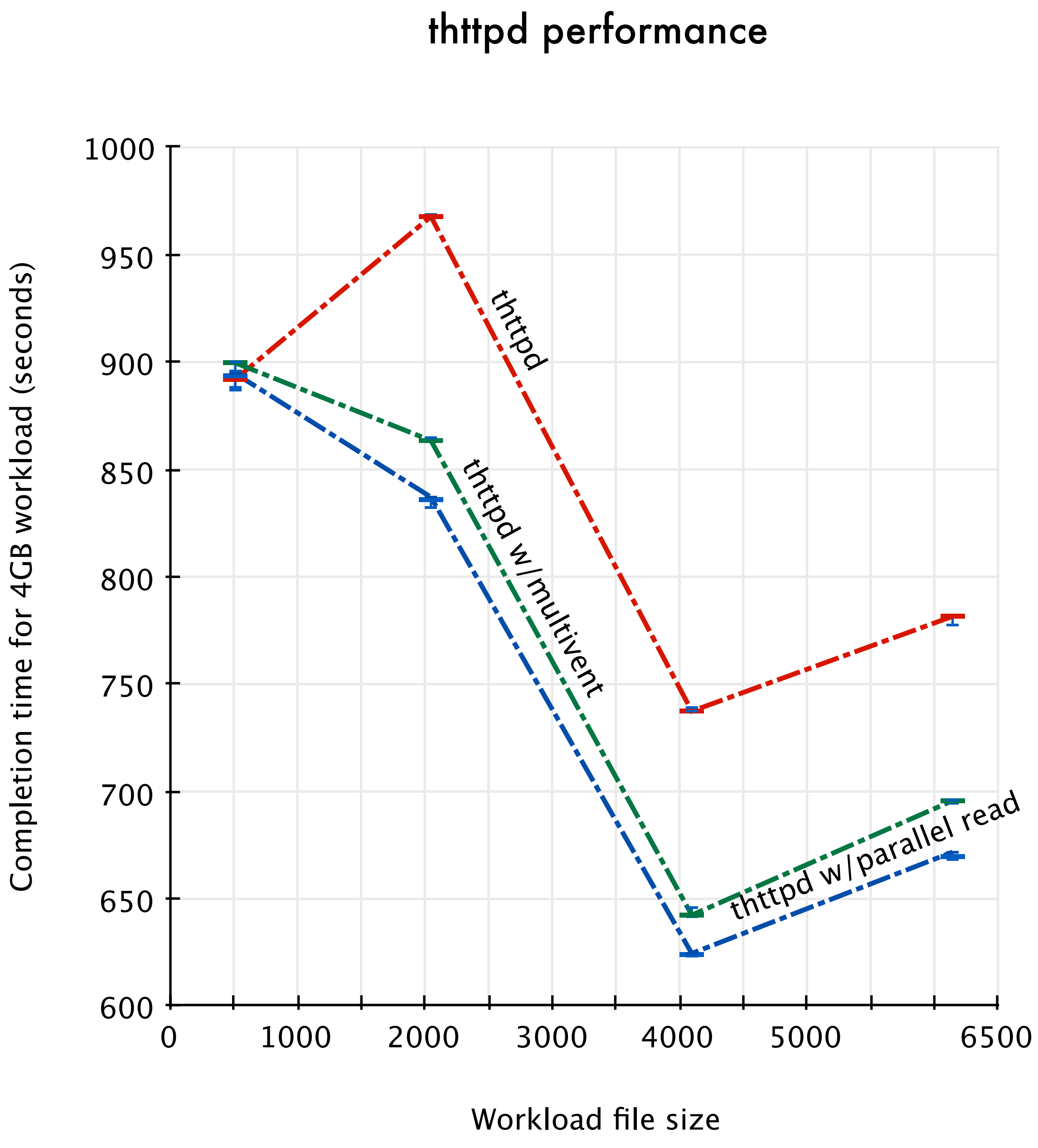


Figure 6.1: Time to fetch 4GB of data from `thttpd` over different file sizes.

6.4.1 Motivation

`tor` uses TLS 1.0 to establish virtual circuits to carry user traffic. The TLS handshake uses asymmetric key encryption, which is very CPU intensive. As `tor` is implemented as an event-driven server, the single thread could get overwhelmed when multiple circuits are being established.

Aware of this, `tor`'s authors have implemented a pool of 'CPU workers' to perform CPU-intensive tasks in multiple threads. This pool is used for performing the TLS handshake when a new circuit is established.

This approach has, in fact, improved `tor`'s performance in setting up circuits as can be seen from the performance testing described in Section 6.4.3.

Unfortunately, even the expert programmers behind a large, complex server such as `tor` can fall victim to the difficulties of multi-threaded programming as can be seen from the following comment in `cpuworker.c`:

```
/** We have a bug that I can't find. Sometimes, very rarely, cpuworkers get
 * stuck in the 'busy' state, even though the cpuworker process thinks of
 * itself as idle. I don't know why. But here's a workaround to kill any
 * cpuworker that's been busy for more than CPUWORKER_BUSY_TIMEOUT.
 */
```

If `tor` were parallelized using `elyze` and `multivent`, the CPU worker pool would not have been necessary, so long as the event handler that performed the CPU-intensive task could be executed concurrently with itself and other event handlers.

Thus the goal with `tor` was to eliminate the CPU worker pool with its associated bugs, but maintain the same level of performance in the establishing of circuits.

6.4.2 `elyzing`

Before analysis: `tor`'s interface to the CPU worker pool is over a UNIX domain stream socket. An event handler makes a request to a CPU worker by calling `assign_onionskin_to_cpuworker()` which writes the request to one end of the connection. When the CPU worker thread finishes processing the request, it writes the response back to the connection, which triggers a `tor` event handler that calls `onionskin_answer()` to deliver the response.

We eliminated the CPU worker threads and their associated communication channels. The new implementation of `assign_onionskin_to_cpuworker()` function delivers the request via session to an event handler: `cpuworker_callback()` that is immediately triggered. The event handler processes the request and delivers the response to another event handler: `cpuworkdone_callback()`, which calls `onionskin_answer()` to complete the request.

We made a few other changes due to some peculiarities in `tor`'s interaction with `libevent`; these had no effect on functionality.

This version of `tor`, running single-threaded with `libevent` shows us (in Section 6.4.3) why the authors introduced the CPU worker thread pool: as the number of parallel circuit requests increase, performance degrades.

Analyzing: `elyze` completes its first iteration over `tor`'s source code in 70 minutes. However, `tor` uses recursion extensively: there are 131 recursive call chains of varying lengths, one of which is 27 calls deep. Furthermore, `tor` has over 6000 globals, with up to 3200 being used in each of the two main event handlers.

Due to these factors, `elyze` takes a total of 18 hours to completely process `tor`'s call graph. There are relatively straightforward engineering solutions to improve this performance, such as caching and re-validating results from previous runs, which have been left for future work.

`elyze` reported that `cpuworker_callback()` conflicted with itself. This would prevent the event handler from being invoked by multiple threads concurrently, which was essential to matching the original `tor`'s performance.

The reported conflicts related primarily to statistics updates; these were in `note_crypto_pk_op()` in `rephist.c`. We replaced all these with atomic operations which are recognized as such by `elyze`; this successfully eliminated the constraint.

6.4.3 Results

Tests were carried out on a private `tor` network, composed of five routers. `tor`'s control protocol was used to establish control connections to each of these routers. Every test requested each router to establish multiple three-hop circuits with the other routers; the time taken to establish each circuit was recorded. We ran tests on the following versions of the server:

1. Unmodified `tor`

2. `tor` without CPU worker threads, using `libevent`

3. `tor` without CPU worker threads, using `multivent`

Figure 6.2 shows the results of these tests. We see that as the number of parallel circuit requests increases, the average time required to establish a circuit increases. The single-threaded version of `tor` suffers from the CPU-intensive nature of circuit establishment.

`tor` with `multivent` actually out-performs `tor` with the CPU worker thread pool. We attribute this primarily to the reduction in thread communication overhead (the original `tor` uses a UNIX domain stream socket), and to the removal of the culling mechanism that the `tor` authors used to work around their threading bug.

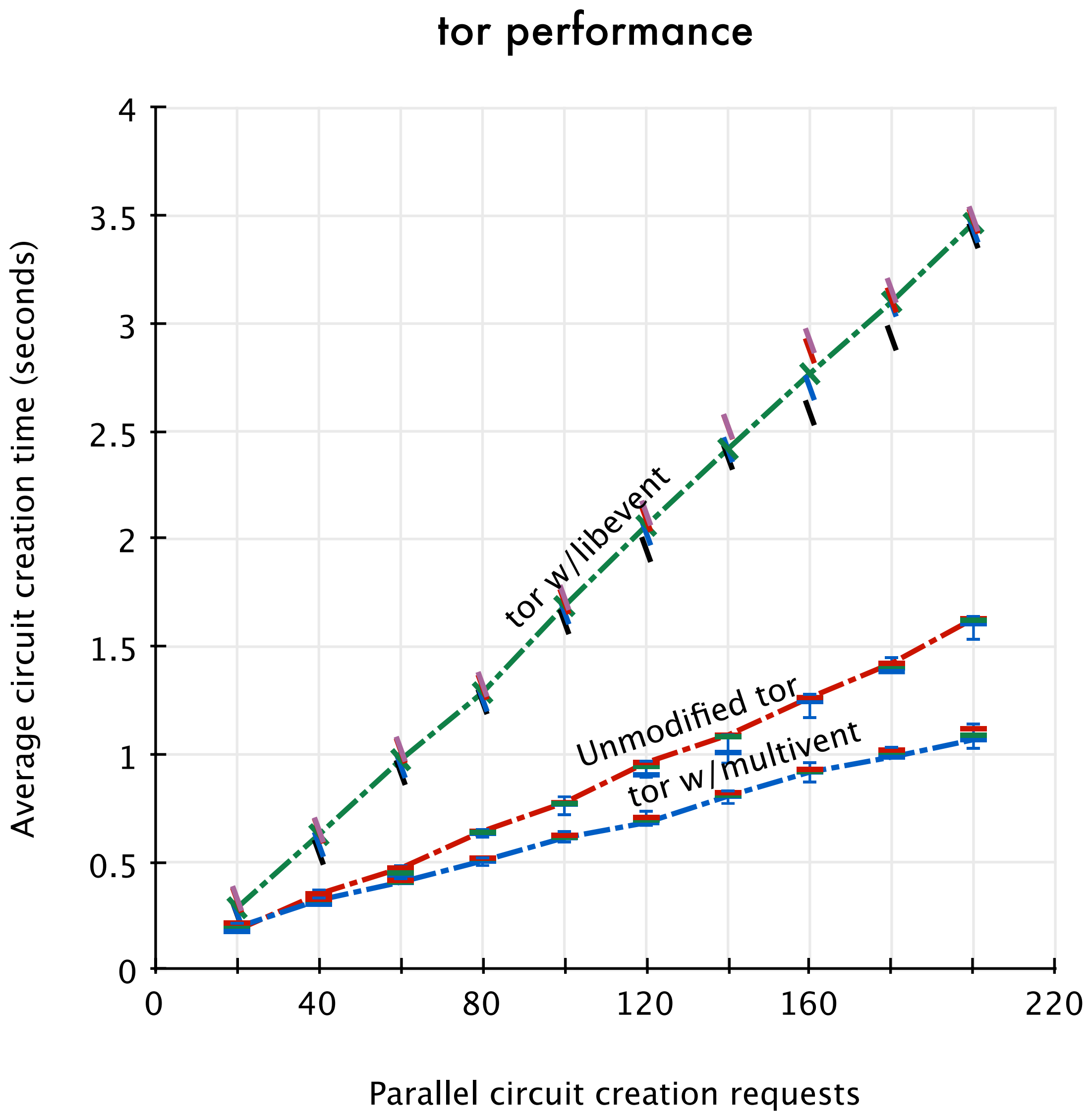


Figure 6.2: Average time to establish a `tor` circuit, over number of parallel circuit requests.

Chapter 7

Conclusions and Future Work

7.1 Conclusions and Future Work

Parallel programming should be safe by default. Just as garbage collection eliminates an entire class of bugs relating to memory management, safe parallelism ensures that data races between threads cannot occur. This allows programmers to focus their efforts on improving performance rather than on attempting to reproduce and locate subtle bugs, which will result not only in more reliable applications, but also in increased programmer productivity.

`elyze` and `multivent` and their application to real world server applications as presented in this dissertation demonstrate that safe parallelism is achievable on existing code, that the system guides the programmer in increasing opportunities for concurrent execution, and that incremental performance gains result from development effort.

There are many avenues to extend this work.

7.1.1 OS resources

`elyze` assumes that conflicting objects are blocks of memory. There are, however, other resources that can cause conflicts, such as files and pipes. Recognizing these resources and manipulations to them is necessary for `elyze` to correctly identify all conflicts.

7.1.2 Change impact

During the course of the development of a server application, changes to the code may create new conflicts between event handlers, or remove existing conflicts. Thus, there may be changes to the constraints determined by `elyze` between one analysis and the next. Assessing the impact of such changes and reporting them to the programmer would improve `elyze`'s utility as a software engineering tool.

7.1.3 Runtime enhancements

There are a number of potential enhancements to the runtime scheduler. The current implementation acquires all required locks prior to event handler invocation. Delaying lock acquisition to just prior to an access could increase concurrency as some conflicting accesses could occur only down certain conditional branches; if those branches were not taken, no conflict would exist.

7.1.4 Others

- Support for other languages, such as Java.
- Evaluating the feasibility and utility of safe parallelism for applications other than servers, and on platforms such as Android.
- Exploring the application of safe parallelism to programs written to different models.

Bibliography

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen (DIKU report 94/19), May 1994.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

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

[4] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Not.*, 35(5):35–46, 2000.

[6] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241, 1994.

[7] H.S. Kim E.M. Nystrom and W.M.W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. 11th Static Analysis Symposium*, August 2004.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, 1994.

[9] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 291–296, New York, NY, USA, 2007. ACM.

[10] 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.

[11] Cormac Flanagan and S.N. Freund. Automatic synchronization correction. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

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

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

[14] Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Efficient workstealing for multicore event-driven systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 516–525, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[16] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[17] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[18] http_load. `http://www.acme.com/software/http_load`.

[19] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

[20] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM.

[21] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. *SIGOPS Oper. Syst. Rev.*, 44:93–101, August 2010.

[22] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[23] Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. Quasi-static scheduling for safe futures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 23–32, 2008.

[24] 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.

[25] J.K. Ousterhout. Why threads are a bad idea (for most purposes). In *Invited talk at the 1996 USENIX Conference*, January 1996.

[26] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[27] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

[28] Niels Provos. Libevent – an asynchronous event notification library, 2000. `http://www.monkey.org/ provos/libevent/`.

[29] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[30] 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.

[31] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 59–66, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[33] H. Sutter. The trouble with locks. 2005.

[34] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. Back to the futures: incremental parallelization of existing sequential runtime systems. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 583–597, New York, NY, USA, 2010. ACM.

[35] thttpd - tiny/turbo/throttling http server. `http://www.acme.com/software/thttpd`.

[36] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, October 2001.

[37] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 265–274, New York, NY, USA, 2008. ACM.

[38] 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.

[39] 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.