Katana: A Specialized Framework for Reliable Web Servers

Ajeet Shankar
aj@cs.berkeley.edu
UC Berkeley

Bill McCloskey
billm@cs.berkeley.edu
UC Berkeley

Abstract

E-commerce server reliability is critical, as downtimes cost an average of $10,000 per minute [40]. Commercial web server development today is done with fairly generic programming languages, like Java, Perl, and C#. The generality of these languages, while permitting a wide range of target applications, makes it difficult to guarantee reliability: dynamic type errors, race conditions, and resource leaks contribute to instability. Though the languages may detect such errors at runtime, the resulting downtimes in production code are costly.

We present Katana, a specialized framework for creating reliable web servers. Generality is exchanged for specific capabilities tailored to server operation. In particular, servers written with Katana benefit from these properties: truly statically type-checked code; specialized language features for common server tasks, such as data transformation and formatted output; native, statically-checked database interaction; automatic memory management and concurrency control; and built-in state-sharing mechanisms. By eliminating much of the complexity inherent in general-purpose frameworks and unnecessary for web server operation, while retaining a suitable range of expressiveness, Katana servers are not subject to several entire classes of bugs that plague existing web servers, and are thus more reliable.

Preliminary results indicate that Katana is comparable to existing server frameworks in terms of ease of use and performance, suggesting that it is a viable architecture for real-world web servers.

1 Introduction

Over the past ten years, Internet servers have grown tremendously in importance. Web servers have become the public face of the Internet: as of January 2006, seventy-five million of them crowd the network [33]. Web mail has also become important, with enormous services like Hotmail, Gmail, and Yahoo! Mail hosting hundreds of millions of users and contributing to the 31 billion emails sent daily [41]. E-commerce continues to grow by billions of dollars annually [11]. The overriding theme in this story is that dynamic web sites have become the rule rather than the exception. To meet the demand, web servers have become increasingly powerful. Most servers are now merely containers for programs written in general-purpose languages like Perl, Python, Java (via J2EE [21]), and C# (via .NET [10]). Although these languages are very expressive, they provide insufficient reliability guarantees. For instance, the lack of static typing in Perl and Python can lead to runtime errors that escape QA testing and only manifest themselves during production. Java and C# are statically typed, but they still have many potential runtime errors, as well as race conditions, deadlocks, and resource leaks.

These programming frameworks do provide a basic level of safety in that when programs fail unexpectedly, they terminate rather than misbehave. However unexpected termination, while safe, is still unreliable: a server that has gone down cannot complete a sale. This lack of reliability can be deadly; according to the Gartner Group, e-commerce downtimes cost an average of $10,000 per minute [40], in addition to intangibles such as a loss of reputation and increased customer irritation.

Why are such frameworks unreliable, even given this large cost? The problem lies in the generality of the framework design. A general programming framework, such as .NET, is by nature very powerful. This power renders such frameworks attractive to server designers, since high performance servers are generally complicated beasts. E-commerce servers, for instance, need to handle a variety of tasks, from processing a very large volume of user requests, to executing server-side logic in the business of displaying pages and constructing orders, to accessing a back-end database for product information.

However, there is a critical flaw with using general-purpose frameworks: their power, which allows servers to handle each of these critical tasks, is manifested in a set of generic features that, if the framework is truly general, must be suitable for not just for web servers but for all kinds of other applications as well. This generality results in a framework that is suboptimal for the specific domain of reliable web server development. The results, naturally, are less reliable web servers.

Katana: a solution specialized for reliability. We introduce Katana, a new approach to server development that directly attacks this problem of unreliability via specialization. Katana is a server framework, like J2EE or .NET, but one designed specifically for web server reliability. It is based on the philosophy that, in the context of web servers, the flexibility offered by general-purpose languages like C and Java is rarely useful and often quite dangerous. Katana eliminates features like unrestricted threaded programming, reflection, dynamic typing, and garbage collection and replaces them with ones that are safer, faster, easier to use, and better suited to server development.

This concept of specialization is the heart of the Katana philosophy. Web servers tend to have particular and consistent execution behavior: they field many requests (usually independently from one another), communicate with a persistent state mechanism such as a database or session store, manipulate and transform data, and generate formatted output such as HTML. Because we know that web servers follow this restricted set of actions, we can exchange the generality of existing frameworks for the specialization of a framework finely tuned for servers, focusing especially on reliability. Some representative aspects of the Katana framework are:

- A statically type-safe language
- A restricted yet suitably expressive language feature set
- Statically checked database interaction
- Automatic region-based memory management
- Built-in concurrency with controlled state sharing

Each of these aspects, discussed in detail in this paper, is tailored towards server reliability. In this way, Katana drastically reduces the number of avenues by which bugs can be introduced into web servers.

Practical concerns: ease of use and performance. Two concerns of any server framework designed for reliability are whether it is expressive and easy enough to use to allow for the creation of real web servers, and whether such servers are fast enough to be practical in a production environment.[1] While these two goals are not the main focus of this paper, we realize that they must be met to some extent if any system purporting reliability is to be taken seriously in the real world.

Preliminary evidence suggests that Katana does indeed meet these two goals. The philosophy of specialization that is the basis for Katana's claims to reliability is crucial here as well. The design of the Katana language, K, reflects not only the overarching aim of reliability, but also a focus on ease of use. Common server logic idioms, such as structural pattern matching, data structure iteration and transformation, regular expressions, structured output (like HTML) and so on, are built directly into the language. Preliminary experiments with two medium-sized test cases indicate that programmer ease of use with Katana, measured in lines of code and development time, compare favorably against existing solutions such as J2EE and even Perl. Further details are presented in Section 5.2.

As with ease of use, specialization aids crucially in the area of performance. In Katana, common-case web server execution behavior is used to motivate specialized performance-oriented features in the same way it is used to increase reliability. For instance, a cooperative threading model and region-based memory management exploit the relatively short processing time of most web requests to reduce overhead. The domain-specific language K makes program semantics more evident to the compiler by focusing expressiveness on the particular domain of web servers. Thus, it enables several language-based optimizations unavailable to more general purpose languages. Preliminary benchmarks show that Katana performs extremely favorably against the J2EE dynamic server framework, and comparably to even an optimized static web server. More details can be found in Section 5.3.

Katana as a long-term platform. One final concern is the tendency of any platform to exhibit "feature bloat" as it matures and evolves to better fit its target domain. For a specialized framework such as Katana, such bloat, if improperly implemented, can be devastating, adding the complexity and unreliability that the framework was expressly created to avoid.

To combat this problem, care was taken in the design of Katana to ensure that any modifications to the framework could be made in a "native" fashion, retaining the domain-specificity that is crucial to its success. Thus, the Katana compiler for K is completely extensible; any new language features can be integrated directly into the language rather than being hacked in on top of it. Further-

[1] Of course, these concerns also plague existing, general-purpose frameworks: time and skill are often required to make a program perform well when it uses garbage collection, interpretation, or just-in-time compilation, as many of these frameworks do.

more, the Katana runtime is also extensible: new threading and memory management modules can be swapped in. In this way, if a given class of servers would perform faster or more reliability with a particular runtime environment, it is not necessary to “generalize” the Katana runtime to provide that environment; instead it can be plugged in independently without having to alter the default runtime at all.

Paper contributions. The primary contribution of this paper is the Katana framework for web server creation specialized for reliability. The static and runtime portions of the framework are described in Sections 3 and 4, respectively. Additionally, we present a full implementation of the Katana system and preliminarily assess its real-world viability by examining its ease of use and performance in Section 5. Finally, we discuss related work and conclude. To begin, however, we describe a simple example server.

2 Example Server

The best way to illustrate how Katana works is by example. Figure 1 shows the architecture of a simple web server that returns a list of employees from a database. Katana code is written in the language K, described in Section 3, and is structured into modules. Modules are compiled to C and linked with the runtime and libraries. In this example there are two modules. The first module dispatches requests and fetches results from the database. The second module handles formatting of output data. Either of these modules may make use of Katana library functions.

The example contains three functions in K that are written by the user (they are shown at the left of the figure). The first one decides, based on the URL, how the request should be processed. It uses pattern matching with regular expressions to parse the URL, although the Katana support libraries also include more sophisticated URL parsing capabilities. To fetch a list of employees, the second function begins a transaction and performs a database query. It returns the list of records resulting from the query (list types are delimited by square brackets in our syntax). The third function formats the employee list in HTML.

Regular expressions, statically checked database access, and automatic string manipulation and interpolation are all features of Katana. Efficient automatic memory management and threading are built into the Katana runtime. All together, this simple example requires only a few dozen lines of code together with some supporting libraries provided by Katana. This modest investment yields a statically type-safe application server free of most common runtime errors that scales to thousands of connections per second (ignoring database overhead).

3 Katana’s Static Framework

Both the Katana static language framework and its runtime are specialized for reliability. In this section we describe the features of the static framework. The heart of this framework is K, the domain-specific language in which Katana servers are written. We start by describing the basic features of K, and then delve further into some of its significant specializations.

3.1 The Basics of K

In repsonding to requests, web servers commonly perform several tasks:

1. Data transformation and aggregation work
2. Control tasks, like dispatching on a URL or other input
3. Access to persistent data like session state or a database
4. Generation of formatted output (usually HTML)

The language K is specialized to make the completion of these tasks easy and safe. Critically, ensuring that tasks are easy to complete is as valuable as making sure that individual operations are reliable: even a perfectly safe language can lead to algorithmic bugs induced by unnecessary complexity unless it provides the programmer with a set of expressive tools.

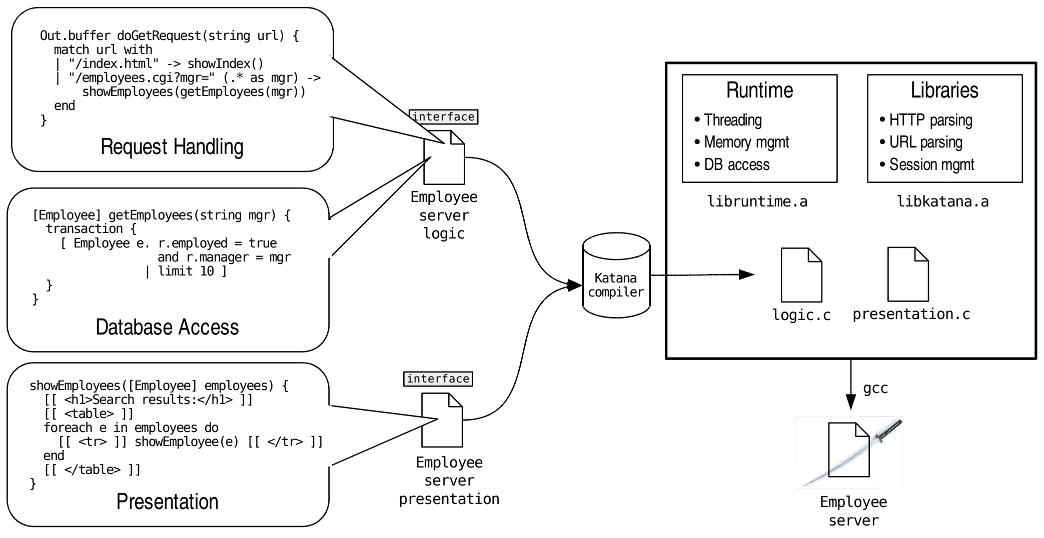
K enables the former two tasks via an expressive type system and structural pattern matching.

In addition to the standard base types and references, K supports a variety of aggregate types for easy data manipulation. Figure 2 shows some examples. There are no pointers in K, only (non-null) references, eliminating a large class of safety errors. Our usability studies (see Section 5.2) indicate that these are more than adequate for the server domain.

Server logic is often data-driven: it must be transformed, analyzed, or acted upon. Thus, K supports structural pattern-matching, which makes it easy to decompose data structures into their components. See Figure 3 for examples. We have found pattern matching, which we have extended with regular expression support, to be an enormously useful and concise tool in programming servers.

K is strongly typed at compile time, negating the need for dynamic safety checks[2]. Static typing ensures at compile-time that parametric types like lists are used properly, in contrast with languages in which proper usage of generic container types is only assured at runtime.

[2]In fact, the type system is structured to allow for full type inference, although we did not implement an inferencer.



```
Out.buffer doGetRequest(string url) {
  match url with
  | "/index.html" -> showIndex()
  | "/employees.cgi?mgr=" (.* as mgr) ->
      showEmployees(getEmployees(mgr))
  end
}
```

```
[Employee] getEmployees(string mgr) {
  transaction {
    [ Employee e. r.employed = true
                  and r.manager = mgr
                | limit 10 ]
  }
}
```

```
showEmployees([Employee] employees) {
  [[ <h1>Search results:</h1> ]]
  [[ <table> ]]
  foreach e in employees do
    [[ <tr> ]] showEmployee(e) [[ </tr> ]]
  end
  [[ </table> ]]
}
```

Figure 1: An example Katana server, including the complete server infrastructure. This server either displays a welcome page or returns the list of employees under a particular manager. The employee list is read from a database.

```
type SessionId = int;
type Story = { Author a, string text };
type User =
 | RegisteredUser(string, string)    // username, password
 | Guest
end;
type AssocList[t] = [ (string, t) ]; // list of pairs
```

Figure 2: Sample types in the K type system, including base types, records, unions, tuples, lists, and parametric types.

Katana compilers are portable to any platform with a C compiler. Programs in K themselves are platform independent.

Memory management in K is automatic, stamping out a major class of errors. A danger of automatic memory management is that it can bloat the heap or cause programs to pause at arbitrary times; in Section 4.2, we argue that Katana's memory management scheme, specialized to the server domain, suffers from neither of these problems.

The third web server task, shared state, is often the source of many concurrency and memory bugs, as dangling pointers, race conditions, and deadlocks can occur when trying to access concurrently available data. Rather than allowing the server programmer to manage shared and persistent state in an arbitrary fashion, K exposes the two most common state-storing mechanisms, session stores and databases, in a structured, type-safe way. K also has a specialized solution for the fourth task, generating output. Details of these language extensions are below.

```
bool isLoggedIn(User u) {
  match u with
  | RegisteredUser(uname, pass) -> getPassword(uname) = pass
  | Guest -> false
  end
}

Cart actUpon(Cart cart, Item item, string action) {
  match action with
  | "purchase" -> cart := addItem(cart, item)
  | "remove"   -> cart := deleteItem(cart, item)
  | _          -> ()      // anything else
  end;
  cart
}

(string,string) getCookiePair(string cookie) {
  match cookie with
  | /(.* as name) "=" (.* as val)/ -> (name, val)
  | _ -> ("","")
  end
}
```

Figure 3: Pattern matching with unions, strings, and regular expressions.

3.1.1 The Module System

A Katana server is constructed from modules. Each module communicates with other modules using a common interface system that supports abstract types for modularity. The module system is also a convenient way to interface with C code—any module can be written in C as long as it has a Katana interface file. Interestingly, since Katana compiles to C, it is usually just as efficient to write code in K instead of C, and much easier.

Katana also provides a number of useful libraries via the module system, some written in K and some in C, including HTTP header parsing (with cookie support), input and output buffers, and URL parsing and manipulation.

```
type Cart = [ (Item,int) ]; // list of item, quantity pairs
store[Cart] carts;          // defines the Cart store

Cart getCart(string sessionid) {
  match Session.get[Cart](carts, sessionid) with
  | Some(cart) -> cart      // user already has a cart
  | None       -> []        // empty cart
  end
}
```

Figure 4: Session code to manage shopping carts in Java Pet Store, a sample e-commerce application.

```
type Comment = { User creator, bool is_public,
                 Ref[Article] article, string body };

[Comment] fetchCommentsByUser(User u) {
  transaction {
    [ Comment c . c.creator = u and c.is_public = true
                | limit 12, orderby c.date ]
  }
}

[User] fetchArticleAuthors([Comment] clist) {
  [User] res = [];  // empty list
  transaction {
    foreach c in clist do
      if (c.is_public) then
        // automatically fetch article from the database
        // and prepend it to the list
        res := (@ c.article).author :: res
      else
        ()
    end
  };
  res
}
```

Figure 5: Sample database interaction in NewsDog, an online news community. In the first function, `creator` is automatically fetched, while `article` is kept in a lazy reference; a list of `Comment`s is returned. In the second, a `Ref` is dereferenced.

3.2 Session Support

Inter-request session state is very common among dynamic content servers, whether to keep login information or to store items in a shopping cart. A K language feature allows the programmer to store arbitrary session data in a type-safe and thread-safe way. Figure 4 has an example. Naturally, multiple session stores can be created.

All concurrency and memory management issues regarding session support are handled automatically by the Katana runtime, as detailed in Section 4.

3.3 Database Support

The de facto method of data storage and retrieval in modern Internet servers is the relational database. Unfortunately, there are several problems with traditional server/database interaction via SQL.

Hierarchical data that is conceptually grouped together (for instance, the fields of a data structure) must be marshalled to and from the database for each transaction, often via hand-written custom marshallers or a tedious process of reconciling the database schema with the object layout to the application server.

Furthermore, it is difficult to generate queries that interact with the database in a provably safe way. SQL statements are essentially untyped; the validity of a statement with respect to its parameters and the tables it affects can often only be determined at runtime. Also, query construction is usually done by concatenating strings (and dealing with the associated quoting and security issues involved with inserting external data into a SQL command); even with the bind capabilities available in newer databases, which allow programmers to abstract the parameters away from the actual string representation of a query, the correctness of a query is still difficult to determine.

Additionally, there is rarely a consistent way to handle references to other records in memory (“swizzling” [32]). Null pointer dereferences and more devious errors are common when there is no consistent semantics for storing and restoring references.

Katana explicitly addresses these problems at the language level. We have integrated a database interface into K that provides:

- A direct relation between user data structures and tables in the database
- Automatic, safe, and efficient marshalling of data structures to and from the database
- Statically type-checked query construction
- A clear, flexible semantics for handling references

The programmer specifies which, if any, of his named types are to be stored in the database with the `dbtype` declaration. Each `dbtype` corresponds to a table in the database, and each field a column; in all other respects it is identical to a normal type. `dbtype`s can contain other `dbtype`s, in which case the contained objects will be fetched eagerly from the database when the parent object is. `dbtype`s can also contain a lazier reference, `Ref`, to other `dbtype`s; dereferencing a `Ref` automatically fetches its associated object from the database. These semantics eliminate dangling pointer problems and allow the programmer to control the kind of swizzling—eager or lazy—on a per-reference basis. See Figure 5 for an example.

All database interactions in Katana occur inside a `transaction` scope. The boolean expression in the query, analogous to the `where` clause in SQL, can be arbitrarily complex and is type-checked with respect to the database schema at compile time. The query is then compiled down to an abstract syntax tree (AST) from which a SQL query is generated.[3] Thus, the query system verifies

[3]Katana also exposes this underlying AST to the server writer for the (compile-time type-checked) programmatic construction of queries at runtime. However, queries that can be entirely constructed at compile time are.

```
presentArticlePage(ArticlePage apg) {
  pageTop(apg.username)
  presentArticle(apg.article, apg.id)

  string s = if apg.numComments = 1 then "" else "s"
  [[ <hr noshade><div align="center">
     $apg.numComments Comment$s</div> ]]

  foreach c in apg.comments do
    [[ <p> ]] presentComment(c) [[ </p> ]]
  end

  pageBottom()
}
```

Figure 6: Presentation code for displaying a NewsDog article.

at compile time via type-checking that query statements are valid, while simultaneously circumventing quoting and safety issues.

Currently, the runtime portion of the DBI communicates with MySQL, although there is no reason why other databases cannot be transparently supported as well.

We feel that the benefits gained by specializing database access at the language level—static checking, safe pointer management, and independence from any particular database or strain of SQL—outweigh the moderate increase in expressiveness that comes with ad-hoc user-constructed queries. We have evaluated the database support in the context of two case studies (see Section 5) and have found it expressive enough to handle those tasks.

3.4 Presentation Mode

Web servers generally need to send data back to a requesting client, and this data tends to be formatted in a language such as HTML. Much as existing frameworks have custom solutions for this problem, such as JSP and ASP, K has a *presentation mode* that makes generating formatted output easy. The fundamental data type in this mode is the string: strings are generated, concatenated, and interpolated.

Using traditional languages to generate lots of string content is usually quite painful; instead, K's presentation mode makes strings the default data type. Expressions evaluate to strings, and functions simply are a sequence of string expressions. Also, as iteration over data types is very common in creating output, natural iteration mechanisms are provided. The combination of these features allows for very concise string manipulation. The division of formatting components into functions rather than files makes reusing code and constructing complete pages from components natural.

Furthermore, as in JSP/ASP and Perl, code can be interspersed with strings, and string constants support variable interpolation. Of course, unlike in Perl, all functions are statically type-checked, so no dynamic errors can arise.

Figure 6 shows sample presentation code.

Since K is designed so that strings are immutable, an opportunity for optimization arises. Repeated concatenation of strings as the output is aggregated does not occur; instead, the output is accumulated in a list of string references, and relayed back to the client in optimally-sized chunks. No copies are ever made.

3.5 New Languages and Features

The initial Katana implementation relies on these features of K, which we believe are useful for application and content servers. However, Katana also makes it easy to add new language extensions and even new languages in order to increase expressiveness as necessary. Naturally, any modifications to K must take care to not violate its safety properties.

The entire compilation infrastructure that translates K to C is extensible. We supply libraries for parsing, type checking, and code generation of the base features detailed above. A language designer need only describe those additional features that are unique to his extension or new language. Parsing is done with Elkhound [29], a GLR parser generator that supports user-controlled resolution of parsing ambiguities. Type checking and code generation can be extended by a mechanism similar to object-oriented inheritance.

Providing incremental extensibility at every level makes it easy to apply a variety of modifications. For example, adding syntactic sugar to K requires only an incremental change to the parser; exploiting an optimization opportunity can be done at the code generation level without any other work. Of course, larger-scale design changes are also possible.

An additional benefit of incremental extensibility is that, if a new language is required, it tends to share a common syntax and semantics with K and only differs in ways that make it more naturally suited to its chosen domain. Code in different languages can interact seamlessly as long as they expose Katana interface files and share the common binary interface used by the K compiler.

We relate our experience adding several new features to K via these extensibility mechanisms in Sections 5.2.3 and 5.2.4.

4 Katana's Runtime Framework

Katana servers use a runtime that has been specialized for web server behavior. This section describes the most important features of the runtime.

4.1 Concurrency

In designing Katana, we needed to decide between a threaded programming model and an event-driven model. Although the event-driven model historically has performed better [45], it is more difficult for programmers to understand since control-flow is divided across a set of disparate event handlers. Threads are easier to understand, but many threads packages perform poorly.

We believe that an intuitive programming model is crucial to achieving Katana's goal of reliability. A good model makes writing code easier and simplier, and thus is less likely to introduce bugs. Since threads mimic a system with no concurrency, it is easier to make them transparent to programmers. Recent work such as Capriccio [44] has revived the idea that threads are as fast and as scalable a model as events [25]. Therefore, we designed a cooperative threading library for Katana. This library adheres to the $m{:}n$ model, in which a set of user threads is multiplexed over a set of kernel threads. We expect users to run one kernel thread per CPU.

A standard criticism of threads is that they introduce a new class of errors that are caused by improper synchronization. In fact, though, a cooperative threading model provides exactly the same atomicity guarantee as an event-driven one: that all accesses to shared state between yield points will be atomic. Additionally, no model can avoid synchronization when a programmer chooses to use multiple processors.

In light of these facts, Katana supports a number of mechanisms for shared state, but they are carefully controlled. All synchronization is done automatically for the user so that deadlocks and race conditions never occur.[4] To ensure correctness without compromising performance or expressiveness, we believe that shared state should be domain-specific. As an example, we describe our implementation of session state.

Session state must be shared across connections, but generally it is accessed by only one user at a time. Since there is no actual sharing, there is no need for the session state to be updated. Instead, in Katana, many versions of a user's session state are kept at the server. When a user changes the state, a new version is created. A cookie in the user's browser keeps track of the current version. Because versions are created but never mutated, synchronization problems like race conditions disappear. In this way, the Katana session state design preserves safety and performance, at a small cost in expressiveness.

For other domains, more complex techniques are necessary. The standard database programming model ensures correctness using transactions, two-phase locking, and deadlock detection. Katana's use of domain-specific shared state management allows it to provide database-like semantics where necessary while also supporting a lighter-weight mechanism, like session cookies, when they are a better fit.

[4]The runtime adheres to a locking policy that ensures mutual exclusion and never takes more than one lock at a time.

4.2 Memory Management

The memory access patterns of a web server are consistent, and thus ripe for specialization. Most data is used by only one request handler, and requests tend to be short-lived. Katana uses region-based memory management for this purpose. Each thread is allocated a region, from which memory is doled out incrementally. An allocation costs only as much as a pointer increment. Pointers are never allowed to escape from a region. After a connection ends, all the data allocated by the thread is freed. The memory inefficiency due to the lack of an explicit `free` is mitigated by the short lifetimes of threads in a web server.

Region-based memory management is convenient because it is efficient and completely safe—threads never dereference a freed pointer. Only by specializing for servers, with particular support for session data, can regions be used with so few restrictions. Most region-based systems must deal with inter-region pointers or huge long-lived regions that waste memory. Region-based memory management also has clear advantages over garbage collection, since it never pauses for collection and does not permit memory leaks. These advantages again bolster the reliability of Katana servers, by eliminating another avenue for bugs and complexity.

4.3 New Runtimes

A benefit of making memory management and concurrency transparent to the programmer is that different runtime mechanisms can be plugged in without any changes to the rest of the system or to any of the server's source code. In fact, the Katana runtime API is designed so that runtimes can be linked in even after compilation. The modularity of the runtime system allows Katana to be further tailored to specific server designs or even to different classes of servers. For instance, if a particular application server requires a specific threading model, it can be supported without modifying the rest of the Katana framework. For instance, if Katana were to be re-specialized to POP servers, the runtime might be written to allocate one region per POP command. We have used this feature to debug Katana during development.[5]

[5]In our Katana development environment, we have runtimes that are single-threaded, `pthread`ed, and threaded using our cooperative threads, and of those ones that use `malloc` and ones that use our region-based allocator.

5 Evaluation

In this section we revisit the arguments made about the reliability of Katana servers, and then assess its real-world feasibility in terms of ease of use and performance.

5.1 Reliability

In this paper, we have described the various aspects of the Katana framework and how they have been designed for reliability. We recapitulate those arguments in this subsection.

Reliability is an important metric in the server domain, where five-nines uptime is the industry gold standard. Katana reduces runtime errors and increases stability because its components are designed for web server reliability.

Existing frameworks, like J2EE, guarantee safety in part through dynamic checks. While these checks ensure that unspecified behavior does not occur, they may still raise errors and cause the system or transaction to fail. Often such failures are costly and unacceptable, as runtime errors can require tedious testing to flush out, or worse yet result in an unreliable production system that may only fail when subject to real-world loads.

A key feature of the Katana architecture is strong static typing. All type errors are caught at compile-time, ensuring that the only unsafe actions are algorithmic in nature.

Katana also ensures reliability by restricting and specializing certain language features to eliminate whole classes of errors. For instance, the lack of uncontrolled pointers disallows the possibility of memory corruption. A statically checked, native database interface ensures that common DBI bugs due to erroneous SQL strings cannot occur. Domain-specific management of shared state eliminates race conditions without the need for expensive static analysis or dynamic checks. Another example is resource management: under certain workloads, the Java implementation of Pet Store fails to release database connections and eventually must be restarted [6, 4]—a class of errors that is impossible in Katana, due to automatic resource management.

The results of these efforts are presented in Figure 7.

One final argument for the reliability of Katana servers involves complexity. The language K has been specialized not only for reliability, but also for ease of use (see Section 5.2). Common server operations are built directly into the language as features. The complexity of a roll-your-own concurrency model and shared state mechanism has also been eliminated. The end result is that web servers are easier to write with simpler code. The transparency of this resulting code hopefully makes servers easier to debug and maintain, and thus more reliable.

5.2 Ease of Use and Productivity

As we mentioned in the introduction, a framework like Katana, no matter how reliable, must also be a practical solution if it is to be used in the real world. Early experiments suggest that Katana is both easy to use and fast. In this subsection, we describe ease of use.

We ported one Internet server to the framework and enlisted a developer with no Katana experience to port another. We measure productivity in terms of development time, lines of code, and required programmer skill.

5.2.1 NewsDog

NewsDog [34] is a dynamic web site in which users authenticate themselves and submit news articles and comments. The original NewsDog implementation is written in Perl and runs on Apache, with MySQL as the backend database.

We implemented in Katana a subset of NewsDog, corresponding to roughly 800 lines of Perl in the original implementation. The Katana version took about 6 hours to write and consisted of 573 lines of code. (The above figures include roughly 300 lines of embedded HTML.)

Perl's extreme conciseness lends credence to the argument that the Katana DSLs are appropriately expressive. We were able to use the existing NewsDog database schema without modification for the Katana version. K's presentation mode made it trivial to port the embedded HTML from Perl. The type system proved adept at catching bugs.

5.2.2 Java Pet Store

As a further test of Katana, we enlisted a developer to re-implement Java Pet Store [39], a sample e-commerce application intended to illustrate the functionality of the J2EE platform. The Pet Store server allows clients to sign in and out, browse a catalog of available pets, search for specific pets, add the pets to a shopping cart and purchase them, and keep information in personal accounts. It maintains state through cookies, sessions, and a database back end. Java Pet Store also includes clients for administrators and suppliers that we did not request the developer to re-implement. He also did not implement the localizations to Japanese and Chinese included with Java Pet Store.

The Katana implementation of the Pet Store is in four modules, respectively containing data type declarations, database access code, application logic, and presentation code. The files have a total of 978 lines of source, and an additional 1506 lines of HTML. In contrast, the the relevant parts of the Java Pet Store implementation are spread across at least 100 source files containing more than 5000 semicolons, along with 30 .jsp files containing

Error \ Architecture	Katana	Perl/Apache	J2EE, .NET	C
null pointer errors	**no**	**no**	yes	yes
database query type errors	**no**	yes	yes	yes
dangling pointers	**no**	**no**	**no**	yes
race conditions	**no**	yes	yes	yes
resource leaks	**no**	yes	yes	yes
runtime type errors	**no**	yes	yes	yes
invalid memory accesses	**no**	**no**	**no**	yes

Figure 7: An examination of the types of safety errors that may occur in a web application created in each of several architectures.

a comparable number of lines of HTML. The domain-specific language K clearly contribute to the reduced code footprint; a single declaration of a database type in K roughly corresponds to three source files for a single entity Enterprise Java Bean (EJB) in J2EE.

The Pet Store was implemented by a developer who had no prior experience with the Katana DSLs. The application took approximately 40 hours to develop, including the time to learn the languages but excluding the time to actually design the HTML layout of the pages, which were taken directly from Java Pet Store. The strong static checking and domain-specific constructs of Katana greatly decreased coding time and caught many bugs that would have been painful to debug at runtime. Especially useful was structural pattern matching: over half of the developer's functions employed it. Additionally, the built-in runtime debug support (including a data pretty-printer) made isolating the remaining bugs easier. The automatic session management and cookie support (for logging in and the shopping cart), and database support (for user and product information and orders) made maintaining state easy. Certain language features such as global variables, type inference, and semi-automatic generation of interface declarations would have decreased development time even more; there are no fundamental obstacles to adding these features and they are future work.

This Pet Store implementation is the one we benchmark, without modification, in Section 5.3. Based on its results relative to other implementations of Pet Store, we feel that high performance servers can be written in Katana with little knowledge or time spent in optimization.

5.2.3 Presentation Mode

One of Katana's strengths is the extensibility of its architecture. To increase productivity in a particular server domain, new features can be added. In this subsection and the next, we describe our experience while adding the presentation mode and database support to K using the

```
addto expr [[
  -> l:LSQUARE i:IDENTIFIER COLON t:typ DOT e:expr
        m:opt_qmods RSQUARE
     { QueryExpr(fst i, t, e, m, l) }
  -> l:TRANSACTION LCURLY es:expr_seq RCURLY
     { TransactionExpr(es, l) }
]]
```

Figure 8: Adding a database support: part of the grammar definition file that defines transactions and queries as expressions. Note that the standard `expr` nonterminal is augmented rather than replaced.

extensible compiler. These analyses are meant to gauge the viability of Katana as a productive framework even as domain requirements change.

The presentation mode is unusual in several ways. For instance, there are no expression separators, functions are defined with no return type, and top-level expressions must evaluate only to strings or to nothing. Furthermore, typed string interpolation is supported.

While we naturally had to make modifications to the language parser, we were able to transform the resulting presentation syntax tree to a standard Katana AST without much work. Using the standard AST, we could invoke the existing type-checker and code generator without modification. The incremental changes we made took two people a day and required only 350 lines of additional code. We feel that the resulting language modification is well-suited to its domain.

This result leads us to believe that creating new language extensions for other unique server tasks is not just a theoretical possibility, but a genuine, practical option when developing servers with Katana. By its very nature, a well-designed extension or new language should increase productivity in its domain, and we hope that a repository of such additions will be created and shared.

5.2.4 Database Support

While we realized that database support is essential to any server architecture, we chose to implement it as a language extension to assess the expressiveness of the system. Implementing database support in K required changes to every level of the system: the parser had to be modified to handle `dbtype`s, transaction scopes and the query syntax; the type-checker had to verify that transactions were used properly and that queries typed correctly; the code generator had to construct SQL statements and manage some runtime state; and the runtime itself had to provide the actual low-level database interface, as well as caching.

One pleasant surprise was that we were able to implement `Ref`s and the programmatic query AST interface entirely natively via the parametric polymorphism and union types provided by the Katana type system.

The ability to incrementally change each phase of the system made most modifications easy. We show a sample addition to the parser in Figure 8. Development of the entire database language and runtime was done in stages by two programmers over two days.

Based on this result, we feel that the extensibility of the Katana framework allows it to remain expressive even as it is applied to new server domains or tasks.

5.3 Performance

While server performance is not the focus of this paper, we did examine Katana in relation to existing popular server frameworks to determine whether it can perform competitively. These experiments are by no means comprehensive, but we feel they serve as a proof of concept of Katana's viability. Our goal was to test Katana against traditional web servers and against frameworks like J2EE. For standard web servers, we used a purely static test. We tested the Java PetStore to compare against a web framework.

5.3.1 Benchmark Configuration

All measurements were taken on a 2-way SMP Pentium Xeon 2.4 GHz server with 1 GB of RAM running Linux 2.6.13. Four similarly configured machines were used to generate load. All machines were connected via Gigabit Ethernet.

Traditional servers often impose an additional overhead for serving dynamic data. To eliminate this variable from the experiment, we used a static test. Static pages are served in Katana as if they were dynamic (by generating them from a K program), so the static results represent the maximum overhead for Katana (and the minimum for a traditional server).

The static test was performed against Knot [44], a small and very fast static server. Since Katana is not targeted at serving large files, our test page was a single byte with HTTP headers. We measured the number

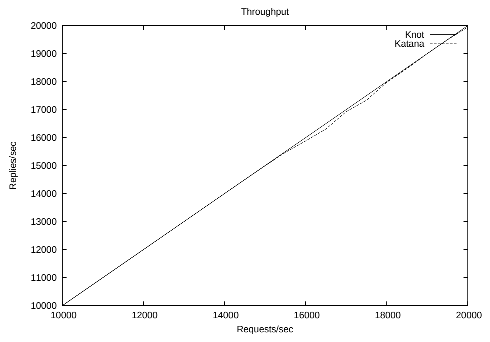


Figure 9: This graph shows the server's ability to deal with load when non-persistent connections are used.

of replies per second that the servers could muster for a given request rate. No persistent connections were used, since they are typically only useful for images and other media.

For the dynamic test, we measured the Katana port of Java Pet Store using the methodology provided in a Middleware Research performance study [8] that compares J2EE and .NET implementations of Java Pet Store. The benchmark simulates a 50/50 mix of users who just browse the site and users who actually purchase items. Each user waits between 2.5 and 7.5 seconds between actions. The number of users is ramped up gradually in increments of 500.

The Middleware tests used an 8-way SMP Pentium Xeon 900Mhz application server with 4 GB of RAM, an identical database server, and 100 clients connected via Gigabit Ethernet. The virtual machines, web server layer, and application server were all heavily tuned; exhaustive details can be found in the research report.

We were unable to reproduce Middleware's results for J2EE and .NET ourselves, as acquiring the appropriate software was prohibitively expensive (and for legal reasons they were unable to reveal which app server they used), and we lacked the expertise to match the enormous amount of performance tuning that their servers underwent. We feel that by matching their workload exactly a reasonable comparison can be made.

5.3.2 Analysis

Figure 9 shows the results of the static benchmark. We were unable to reliably load the servers beyonf 20,000 requests per second. However, this load is sufficient to demonstrate that Katana performs adequately: the number of sites receiving more than 20,000 requests per second (1.2 million per minute) is small. At high load, Katana's performance lags slightly behind that of Knot,

Virtual Users	6000		8000		10000		12000		14000	
Server	RT	WPS	RT	WPS	RT	WPS	RT	WPS	RT	WPS
Katana	1.1 ms	1199.8	1.6 ms	1599.0	25 ms	1981.7	35 ms	2374.2	47.7 ms	2763.1
.NET-C#	5 ms	1196.8	88 ms	1568.3	1476 ms	1531.9	–	–	–	–
J2EE	24 ms	1192.2	330 ms	1498.0	1414 ms	1584.3	–	–	–	–

Figure 10: Response time and web pages per second in the dynamic benchmark.

but only by a few percent. Knot was designed to be extremely efficient; it performs no dynamic memory allocation and it has an extremely specialized HTTP header parser. The fact that Katana competes with Knot using high-level languages and an automatically generated header parser is a testament to the specialization approach.

In the dynamic test, we compared the Katana port of Java Pet Store described in Section 5.2.2 against the J2EE and .NET results provided by the Middleware Company. The results are shown in Figure 10. Katana Pet Store scales far better than the other two implementations; neither of them is able to service more than 10000 clients with an average response time of less than 1.5 s, while Katana scales up to 14000 clients with an average response time of 47.7 ms. We feel that these results demonstrate the advantages of specializing an application framework to a domain like web servers.

6 Related Work

Sun's J2EE [21] and Microsoft's .NET [10] are two popular architectures for creating dynamic web servers. They provide specialized tools for a variety of common tasks, such as database access and output generation. However, much of the server operation is still coded in general purpose languages like Java and C#, and run on general-purpose virtual machines. The problems that arise from this approach—large, complex codebases, suboptimal performance, and too much freedom in dangerous areas like concurrency—are the ones that Katana was designed to fix.

The LINQ language from Microsoft [27], standing for Language Integrated Queries, is an attempt to solve many of the problems of "impedance mismatch" between general-purpose languages and databases. It offers native, type-safe support for SQL and XML queries, as well as several related language features. LINQ is part of the .NET framework. It is still under development.

The Links language [26] is an attempt by the functional language community to solve the same problem. It also encompasses native, type-safe support for database and XML queries, as well as transformation of XML data. It may include continuations as a basis for long-lived web interactions. Also, the designers envision doing client-side and server-side programming together in Links. However, the language is still very much in development.

More mainstream languages like Perl, Python, and Ruby support web programming through libraries. These languages are dynamically typed, which makes them susceptible to runtime type errors. However, it also gives them the flexibility to support many constructs through libraries rather than language extensions. Some of these frameworks do provide "templating languages," similar to the Katana presentation mode. Despite their flexibility, these languages lack static type safety, which also forces them to be interpreted rather than compiled.

Katana's concurrency model draws from the Capriccio thread library [44], which uses cooperative user-space threading to achieve scalability and high performance. Capriccio provides a solid basis for future work in server design, and we are interested in exploring how semantic information derived from DSL compilers could be used by a library like it. The SEDA framework [45] also addresses the problem of highly concurrent servers, although it focuses on the problem of graceful degredation under high load. Although Katana performs well, we are interested in decreasing the response time variance under load using mechanisms like SEDA.

The design of the Katana DSL K is similar to that of many existing languages. The type system and module system draw heavily from OCaml [30, 36], a language that embodies the fact that advanced features do not need to hinder performance. Regular expressions and string interpolation support were inspired by Perl [38], and the presentation language's integration of code and output is akin to that of ASP [1]/JSP [23]. The use of domain-specific languages for systems programming draws from nesC [17] and Click [31], both of which control how C or C++ modules are wired together in a system. nesC also does some additional static checking for properties like atomicity to avoid data races.

In many ways, our specialization approach is similar to that of SPIN [3], which allows applications to augment or replace modules in the operating system, mostly for performance. Like Katana, SPIN takes advantage of modularity and type safety. Other projects, including U–Net [42], application-controlled physical memory [20], and the Exokernel [12] also allow application extensions to the OS, although their interfaces and implementation choices differ. Unfortunately, writing operating system extensions is difficult. We believe that Katana naturally complements these approaches. A specialized Katana runtime could take advantage of OS extensibility without any work from the application programmer. For example, the runtime could perform many of the optimizations used by the Cheetah web server in Exokernel.

Our extensible compiler draws mostly from Polyglot [35], an extensible Java compiler written in Java. Polyglot uses a number of object-oriented mechanisms to allow code and data to be extended simultaneously. Our compiler differs from theirs because it is written in OCaml and uses data constructors and open recursive types to allow code and data extension. The design is similar to Garrigue's work on polymorphic variants in OCaml [15], although we found polymorphic variants themselves to be cumbersome. Our compiler uses the Elkhound GLR parser generator [29] for extensible parsing. In this way, it is similar to the Microsoft C# Research Compiler [19], which uses a GLR parser. However, the C# Research Compiler uses generated visitor classes for extensibility. We expect that this technique would be difficult to use in an incremental development process.

Katana's statically-typed database interface is similar to ObjectStore [24], which integrates database access into C++ code in a type-safe way, and provides a more robust client architecture for caching and complex types. However, it does not have a type-safe programmatic query construction interface, and does not allow the user to control the behavior of pointer swizzling. There have been a number of object-oriented database interfaces that lack type-safe language integration; see [5] for a good summary. Tools such as Microsoft's Fugue [28] and Christensen et al.'s Java analysis [7] attempt to statically verify SQL queries by conservatively analyzing the strings from which they are built. While safe, such techniques can be imprecise and unwieldy compared to native language support.

Memory management in Katana is done using regions. The Apache web server uses its own region system [14]. A number of languages include support for safe regions [9, 13, 16, 18, 22, 43]. Berger et al. [2] demonstrate that most custom memory allocation schemes are not beneficial, except for regions. They present a technique to reduce the memory consumption of region-based systems. Memory consumption is not problematic in Katana, because each regions are attached to threads, which last only short periods of time. Most regions in Katana are only several 4K pages in size.

7 Conclusion

Katana is a server architecture designed for reliability. By virtue of a fully specialized framework, from the language to database interaction to the runtime environment, it eliminates entire classes of bugs common to existing server architectures. Furthermore, early observations indicate that it is easy to use and fast enough for a production environment.

Based on these results, we see Katana as a promising framework for creating high-performance, expressive, and most importantly reliable web servers.

References

[1] Microsoft active server pages .net. `http://www.asp.net/`.

[2] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12. ACM Press, 2002.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283. ACM Press, 1995.

[4] George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *IEEE Workshop on Internet Applications*. San Jose, California, June 2003.

[5] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *The VLDB Journal*, pages 3–14, 1996.

[6] Mike Chen, Emre Kiciman, Eugene Fratkin, Eric Brewer, and Armando Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks*, Washington D.C., 2002.

[7] A. Christensen, A. Mller, and M. Schwartzbach. Precise analysis of string expressions, 2003.

[8] The Middleware Company. J2ee and .net (reloaded): Yet another performance study. `http://www.middlewareresearch.com/endeavors/030730J2EEDOTNET/endeavor.jsp`.

[9] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[10] Microsoft .net. `http://www.microsoft.com/net/`.

[11] United states department of commerce news. `http://www.census.gov/mrts/www/current.html`.

[12] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the SIGPLAN'02 Conference on Programming Language Design and Implementation*, June 2002.

[14] The Apache Software Foundation. The apache http server, 2004. `http://www.apache.org/`.

[15] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000.

[16] David Gay and Alexander Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[17] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM Press, 2003.

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[19] David R. Hanson and Todd A. Proebsting. A research C# compiler. Technical report, Microsoft Research, 2003.

[20] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 187–197. ACM Press, 1992.

[21] Java 2 platform, enterprise edition. `http://java.sun.com/j2ee/`.

[22] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[23] Javaserver pages technology. `http://java.sun.com/products/jsp/`.

[24] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. In *Communications of the ACM 34*, pages 50–63, 1991.

[25] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Operating Systems Review 13*, pages 3–19, 1979.

[26] Links: Linking theory to practice for the web, 2006. `http://homepages.inf.ed.ac.uk/wadler/links/`.

[27] The linq project, 2006. `http://msdn.microsoft.com/netframework/future/linq/`.

[28] Robert Deline Manuel. The fugue protocol checker: Is your software baroque?

[29] Scott McPeak and George Necula. Elkhound: A fast, practical glr parser generator. In *Compiler Construction*, pages 73–88, 2004.

[30] Robin Milner. A proposal for standard ml. In *Polymorphism I.3*, December 1983.

[31] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[32] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. In *IEEE Transactions on Software Engineering*, 1992.

[33] Netcraft web server survey. `http://news.netcraft.com/archives/2006/01/05/january_2006_web_server_survey.html`.

[34] Newsdog. `http://www.newsdog.info/`.

[35] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java. In *12th International Conference on Compiler Construction*, 2003.

[36] Objective caml. `http://www.ocaml.org/`.

[37] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[38] Perl. `http://www.perl.com/`.

[39] Java pet store. `http://java.sun.com/developer/releases/petstore/`.

[40] Tom Pisello and Bill Quirk. How to quantify downtime, 2004. `http://www.networkworld.com/careers/2004/0105man.html`.

[41] Spam statistics 2004. `http://www.spamfilterreview.com/spam-statistics.html`.

[42] Vineet Buch Thorsten von Eicken, Anindya Basu and Werner Vogels. U–net: A user– level network interface for parallel and distributed computing. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*. ACM Press, 1995.

[43] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[44] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for Internet services. In *Symposium on Operating Systems Principles*, 2003.

[45] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles*, 2001.