Katana: A Specialized Framework for Internet Servers

William McCloskey
billm@cs.berkeley.edu
UC Berkeley

Ajeet Shankar
aj@cs.berkeley.edu
UC Berkeley

Abstract

We present Katana, a specialized architecture for creating web application servers using domain-specific languages. Rather than focusing on speed alone, Katana is designed to optimize safety and productivity as well as performance. Servers are written in a set of domain-specific languages (DSLs) that share a common type system, module system, and runtime. We present three DSLs for common server tasks like parsing, request handling, and output formatting. We also describe the Katana language runtime, which allows Katana DSLs to perform well under high load. We evaluate the safety and productivity of Katana via a series of analyses and case studies. Two projects were developed successfully using Katana, one by a developer unfamiliar with the framework. New languages and extensions were also added without difficulty. Finally, we measure performance at serving static and dynamic sites. Katana performs as well as current optimized static web servers, and it exceeds the performance of servers based on J2EE and .NET by a large margin. Katana's response times are lower by two orders of magnitude, and it scales to significantly more clients.

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 May 2004, fifty million of them crowd the network [31]. Email has remained important, with enormous services like Hotmail and Yahoo! Mail hosting hundreds of millions of users and contributing to the 31 billion emails sent daily [38]. 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, and C#. Although these languages are very expressive, they are deficient in other ways. The lack of static typing in Perl and Python is alarming, given the premium that most web sites place on reliability and uptime. Java and C# are statically typed, but they still have many potential runtime errors, as well as race conditions, deadlocks, and resource leaks. Execution is also troublesome in some of these languages. Time and skill are required to make a program perform well when it uses garbage collection, interpretation, or just-in-time compilation.

We introduce Katana, a radical new approach to server development. Katana is a server framework, like J2EE [21] or .NET [10], but one that uses domain-specific languages and a server-specialized runtime to achieve outstanding performance, safe execution, and high programmer productivity. Katana is based on the philosophy that 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.

Currently, Katana provides three domain-specific languages that are targeted to server development. These languages have excellent support for common patterns like parsing of input, database access, session management, and output formatting. Other capabilities are provided by library functions. New features and languages can be added to Katana if the existing ones prove insufficient; the Katana framework is designed to be extensible. Katana languages are built atop a flexible and efficient runtime.

In this paper, we will judge Katana based on three criteria that are crucial to servers.

1. *Safety*, the lack of potential failures[1].
2. *Productivity*, the ease with which a server developer can build a server that meets the desired functional and safety requirements.

[1] We focus on language safety: although security is a critical component of safety, we have not had time to explore it fully.

3. *Performance*, defined in terms of throughput and latency.

All three of these criteria are related because they determine the cost of server design and development. We intend to show that building servers with Katana is first-rate in these three factors—fewest bugs, least labor and skill, and best performance.

1.1 The Katana Way

At the heart of Katana is the idea of specialization. In Katana, servers are written using domain-specific languages (DSLs) rather than general-purpose languages (GPLs) like C# or Java. GPLs offer great power, but they force the programmer to make sacrifices in performance or safety, as shown below. DSLs are restricted languages that are designed for a particular purpose—server development in this case. Some examples of DSLs are the Unix shell, awk, latex, HTML, and Matlab. Perl's text-processing capabilities reveal its domain-specific origin. Domain-specific languages have advantages in four areas that we examine below: expressiveness, ease of use, safety, and performance.

Although they are more restricted than GPLs, DSLs are actually more expressive in their domain. For example, awk clearly would not be useful for serving web pages, but it is much more convenient at processing a text file than C is, since text processing is its specialized domain. Because of their expressiveness, DSLs are more concise and more readable than GPLs, giving them a clear advantage in productivity, even when used by experienced programmers.

DSLs also demand less skill from a programmer than GPLs. Languages like HTML are used by vastly more people than more general ones like Java and C. DSLs are built around a simple set of base concepts that are related to the domain. For example, the core concepts of the Unix shell language are command execution, piping, and redirection—exactly the concepts that a user of the shell must understand anyway. Unlike GPLs, DSLs do not require programmers to know about abstruse features like objects, exceptions, resources, or memory management. Therefore, programmers with few skills can still make use of DSLs to increase their productivity.

DSLs can be safer than GPLs because they lack dangerous features. Languages like Java and C# give the programmer a huge box of tools, including threading, run-time casts, reflection, and explicit management of OS resources. Several of these tools are dangerous if used improperly: threads can have race conditions, reflection and casting may lead to dynamic type errors, and resources can be leaked. DSLs can provide safer domain-specific replacements for many of them. For example, as described below, Katana automatically manages threads, shared data, and resources for the programmer in a safe way.

Since DSLs can make the semantic meaning of a program more evident to the compiler, they can also increase safety through better static checking. For example, Perl can check that regular expressions are syntactically correct at parse-time. In contrast, a Java compiler cannot understand or check regular expressions; it sees them only as strings passed to a library function. DSLs can improve safety by exploiting their greater knowledge of the program semantics.

Performance is one more area in which DSLs can excel. Although high-level languages have traditionally performed worse than low-level ones, the reasons for the difference are not fundamental. Programs get faster when programmers or compilers are able to exploit semantic information about the program. For example, standard compiler optimizations are based on information from data-flow analysis. High-level languages are more difficult to optimize because their behavior is more difficult to understand. For instance, the ability to evaluate arbitrary strings in Perl makes it difficult for the compiler to statically infer information about program behavior. However, unlike traditional high-level languages, DSLs can actually make the semantics of a program more evident to the compiler by focusing expressiveness on a particular domain. Katana is able to make significant runtime optimizations in areas like string manipulation and concurrent data access because of its understanding of server behavior. We believe that DSL compilers offer very promising optimization opportunities, and we exploit some of them in this system.

Although DSLs are important, languages cannot stand alone. They require a runtime to manage memory, concurrency, I/O, and other resources. The Katana philosophy of specialization extends here as well. For example, the Katana runtime uses region-based memory management [40] rather than garbage collection to manage memory. Regions are useful only for certain classes of programs like servers; they are not particularly suitable to a more general runtime—hence the importance of the specialization approach. Katana makes domain-specific optimizations in other areas, too, like concurrency and I/O. In the future, we hope that a specialized runtime could take advantage of extensible operating systems like SPIN [3] and the Exokernel [12].

The contributions of this paper are four-fold. First, we describe a unified language framework specialized to servers, encompassing a type system, a module system, and an extensible compiler. Second, we present three representative DSLs and argue their suitability for the application and content server domain. Third, we describe a runtime environment that is tailored to exploit language features and server behavior to provide high throughput and low latency in the face of heavy load. Finally, we analyze the viability of the Katana framework in terms of safety, productivity, and performance. In this paper, we begin by describing an example Katana server, followed in turn by each of the four contributions. Finally, we discuss related work and conclude.

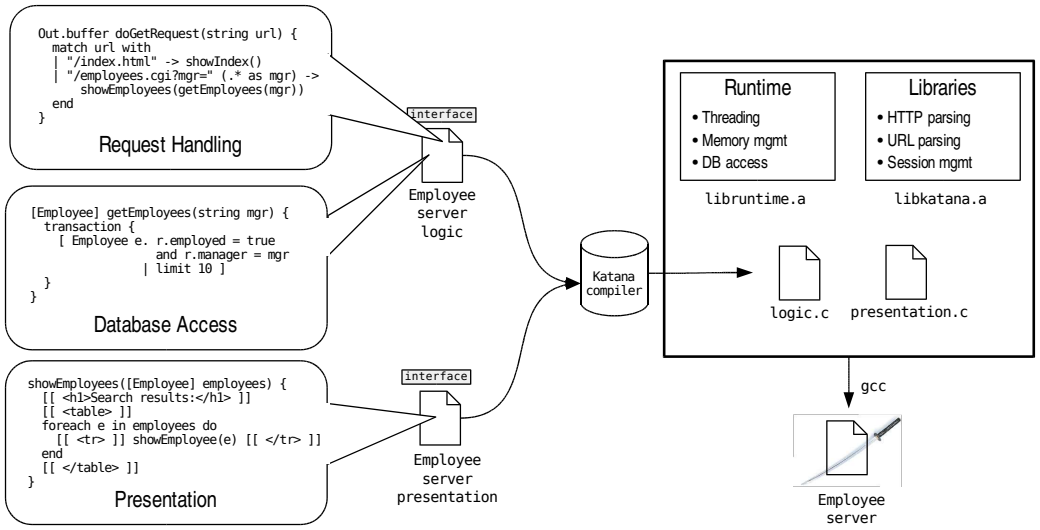

Figure 1: An example Katana server, including the complete server infrastructure. This server either displays a welcome page or returns the list of employees under a particular manager. The employee list is read from a database.

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 structured into modules. Each module is written in one Katana DSL. 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; it is written in our server logic language. The second module handles formatting of output data, and it is written in our presentation language. Either of these modules may make use of Katana library functions.

The example contains three functions 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 these languages. 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. This modest investment yields a statically type-safe application server free of most common runtime errors that scales to 10,000 connections per second (ignoring database overhead).

```
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 Katana type system, including base types, records, unions, tuples, lists, and parametric types.

3 The Language Framework

At the heart of Katana is the unified language framework. Katana provides a number of domain-specific languages (DSLs), each tailored to a common server task and designed with safety, productivity, and performance in mind. To ensure the seamless integration of these languages from all three standpoints, all of the DSLs are based on the same framework, which is detailed in this section.

3.1 Common Features

All Katana languages share a core type system. This design decision facilitates inter-language type-checking and, more importantly, ease of use: the programmer interacts with data structures in the same way, no matter what the language. In addition to the standard base types and references, Katana supports a vari-

```
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.

ety of aggregate types for easy data manipulation. Figure 2 has some examples. There are no pointers in Katana, only (non-null) references, eliminating a large class of safety errors. Our usability studies (see Section 6.2) indicate that these are more than adequate for the server domain.

Katana programs are 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 like Java in which proper usage of generic container types is only assured at runtime.

In addition to the productivity and safety benefits, the unified type system enables a common binary representation so that marshalling data between languages is unnecessary. Katana compilers are portable to any platform with a C compiler. Katana programs themselves are platform independent.

Server logic is often data-driven: it must be transformed, analyzed, or acted upon. Katana languages support 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.

Memory management in all Katana languages 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 5.2, we argue that Katana's memory management, specialized to the server domain, suffers from neither of these problems.

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

3.2 The Module System

A Katana server is constructed from modules. Each module is written in one language and 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 Katana languages instead of C, and much easier.

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

3.3 New Languages and Features

The initial Katana implementation includes three languages that we believe are are useful for application and content servers. Katana also makes it easy to add new languages and extensions in order to increase expressiveness.

The entire compilation infrastructure that translates DSLs 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 language. Parsing is done with Elkhound [27], 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 a language 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 allowed.

An additional benefit of incremental extensibility is that new languages tend to share common syntax and semantics and only differ in ways that make them more naturally suited to their chosen domains. Thus, learning a new Katana DSL is quite easy: the programmer has to learn about only the things that make his life easier.

The three languages detailed in the next section were all created in this manner.

4 Languages for Servers

Server code is not homogeneous; processing is divided into distinct tasks that are more different that than they are alike. The widespread use of presentation languages like JSP demonstrates the importance of language specialization. Katana takes the idea a step further, with a language for each server task, ensuring that

all the languages are similar enough in terms of syntax and semantics that learning and switching between them is easy. Each language is customized to maximize productivity in its given domain without sacrificing safety or performance. In this section we analyze three DSLs that are designed to handle the most common tasks of a content server.

4.1 Tokenizer Language

The first step in any server is request parsing. The tokenizer language is used by Katana libraries to tokenize protocols like HTTP, IMAP, POP, or any XML-based protocol. Although the HTTP parsing library can be shared by most Katana web servers, parsing is an enormously important part of server development, and we expect many servers to require customized parsing[3]. The compiler for the tokenizer language creates table-based lexical analyzers that are specifically designed for tokenizing streaming data from the network. These generated lexers are reentrant and support the Katana data format conventions.

4.2 Server Logic Language

Once a request is parsed, it must be processed. The server logic language is tailored to handle most common 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

The first two items can be accomplished very cleanly using structural pattern matching. For general-purpose logic, the language provides standard control-flow constructs and algebraic expressions. More mechanisms may be added eventually to simplify common idioms, but we have found that most tasks are handled concisely and well. Figures 3, 4, and 5 exhibit code in the server logic language.

The third item, shared state, is the 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, the server logic language exposes the two most common state-storing mechanisms, session stores and databases, in a structured, type-safe way. If desired, new mechanisms can also be added via additional language extensions as in Section 3.3.

4.2.1 Session Support

[3]The Xerces XML parser distributed by the Apache Foundation includes over 1200 sources files and supports several distinct APIs, although it does a good deal more than parsing.

```
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, `author` 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.

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 Katana language extension 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 are handled automatically by the Katana runtime as detailed in Section 5.

4.2.2 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" [30]). Null pointer dereferences and more devious errors are common when there is no consistent semantics for storing and restoring references.

Katana provides addresses these problems at the language level. We have integrated a database interface into the server logic language 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.[4] Thus, the query system verifies 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.

[4]Katana also exposes this underlying AST to the server writer for the 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.

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 6) and have found it expressive enough to handle those tasks.

4.3 Presentation Language

The presentation language addresses the same need as languages like JSP and ASP: output formatting, usually in HTML. The fundamental data type in formatting is the string: strings are generated, concatenated, and interpolated.

Using traditional languages to generate lots of string content is usually quite painful; instead, our presentation language 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 the language is designed so that strings are immutable, an opportunity for optimization arises. Actual concatenation 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.

5 The Runtime Framework

Katana languages are built atop a runtime that has been optimized for application and content server behavior. This section describes the most important features of the runtime.

5.1 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 [42], 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 goals of expressiveness and productivity. A good model saves time in both coding and debugging. Since threads mimic a system with no concurrency, it is easier to make them transparent to programmers. Recent work such as Capriccio [41] 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.[5] 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.

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

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.

5.2 Memory Management

The memory access patterns of a web server are consistent. 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.

5.3 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. 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.[6]

[6]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.

6 Evaluation

In this section we subject Katana to theoretical analysis and a battery of studies to evaluate it with respect to safety, productivity, and performance.

6.1 Safety

Safety is an important metric in the server domain, where five-nines reliability is the industry gold standard. Katana reduces runtime errors and increases reliability by specializing to servers.

Some languages, like Java, 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, and often such failures are costly and unacceptable. This weak form of safety often leads to an increase in the cost of development, 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 safety 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. 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.

6.2 Productivity

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.

One of Katana's strengths is the extensibility of its architecture. To increase productivity in a particular server domain, new features can be added. All of the languages and features presented in Section 4 were created with the extensible framework; in Sections 6.2.3 and 6.2.4, we detail our experiences creating two representative additions.

6.2.1 NewsDog

NewsDog [32] 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. Katana's presentation language made it trivial to port the embedded HTML from Perl. The type system proved adept at catching bugs. We found it easy to switch between DSLs, using the common type system as an anchor point.

We were unable to implement certain aspects of the system, such as authentication (which NewsDog does through encrypted passwords stored in cookies), due solely to time constraints. Others, such as some of the advanced string processing routines, required capabilities Katana did not have, such as a regular expression substitution package in addition to the regular expression matching package we provide. We did not encounter any fundamental limits to Katana's capabilities during this exercise, however; given enough time to flesh out the basic features of the Katana DSLs, we feel that a fairly sophisticated dynamic content server like NewsDog can be written completely in a weekend.

6.2.2 Java Pet Store

As a further test of Katana, we enlisted a developer to re-implement Java Pet Store [37], 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 did 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 a comparable number of lines of HTML. Our domain-specific languages clearly contribute to the reduced code footprint; a single declara-

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 architecture.

tion of a database type in Katana 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; again, 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 6.3. 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.

6.2.3 Presentation Language

In the following two subsections we describe our experience while adding the presentation language and database support to Katana using the extensible compiler. These analyses are meant to gauge the viability of Katana as a productive architecture even when targeted to different domains.

The presentation language is unusual in several ways. For instance, there are no expression separators, functions are defined with no return type, and top-level

```
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.

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 a day and required only 350 lines of additional code. We feel that the resulting DSL is well-suited to its domain.

This result leads us to believe that creating new DSLs 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 DSL should increase productivity in its domain, and we hope that a repository of such DSLs will be created and shared.

6.2.4 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 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.

6.3 Performance

6.3.1 Benchmark Configuration

All measurements were taken on a 2-way SMP Pentium Xeon 2.2 GHz server with 1 GB of RAM running Linux 2.6.5. Three similarly configured machines were used to generate load, with each machine using threads to simulate a large number of clients. All machines were connected via Gigabit Ethernet.

For the static test, we benchmarked a simple web server written with Katana against three web servers. We used version 2.0.49 of the popular Apache [14] server, and compiled it with the `worker` MPM and nonportable atomics for fast thread synchronization. The `worker` MPM was set to a limit of 64 server processes and 64 threads per child; each thread handles a request. Knot [41] is a simple static web server running atop the Capriccio cooperative threads package. Flash [35] is an event-driven static web server that uses a single process to handle requests.

To measure the performance of the servers under heavy load from realistic connection behavior, we use a load pattern similar to the one presented in the SEDA work [42]. Each client makes a request, reads the result and sleeps for 20 milliseconds before requesting the next page. After five such requests are made, the client closes the TCP connection, and then reopens it. To eliminate overhead due to disk I/O, the same 4 KB file is requested each time.

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 (running on two different app servers) 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.

Server	Throughput	Response Time
Knot	498.0 Mbps	46.9 ms
Katana	476.8 Mbps	49.8 ms
Apache	250.0 Mbps	115.8 ms
Flash	318.1 Mbps	127.0 ms

Figure 10: Throughput and latency statistics for all four servers running with 1024 clients.

Middleware also went to great length to ensure that the database was not a bottleneck[7]. The application server was tweaked to do a huge amount of client-side caching, and two database performance consultants spent a week each tuning the database. As we did not have the time (or the knowledge) to do such tuning for the Katana port, we cached all data at the client so that no requests to the database were made.

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.

6.3.2 Analysis

Figure 9 shows the results of the static benchmark. We first examine throughput, on the left. The variance in measurements is quite low, except that the Flash web server performs erratically at 1024 clients. In throughput, Katana and Knot perform about the same. They scale up to 1024 clients with similar throughput. Flash scales well up to 256 clients, but then begins to fluctuate. The fact that Flash cannot server more than 506 connections at once, due to its use of the `select` system call, may explain the problem. Apache exhibits scalability problems at 256 clients. Apache likely experiences difficulties due to its "one kernel thread per connection" concurrency model. With a smaller number of kernel threads, throughput might increase, but latency would become a problem.

Latency measurements at 1024 clients are shown to the right in Figure 9. Average response times for 1024 clients are given in Figure 10. The mean latencies for Katana and Knot are similar, but from the graph, the distributions differ. Knot has a lower median latency but the tail of the latency distribution is longer. The difference is most likely due to subtle differences in the thread schedulers of the two servers. Flash has very

[7] "After our tuning efforts, and before we made any performance measurements that we'd report here, we took great care and did verify that the database was not the bottleneck."

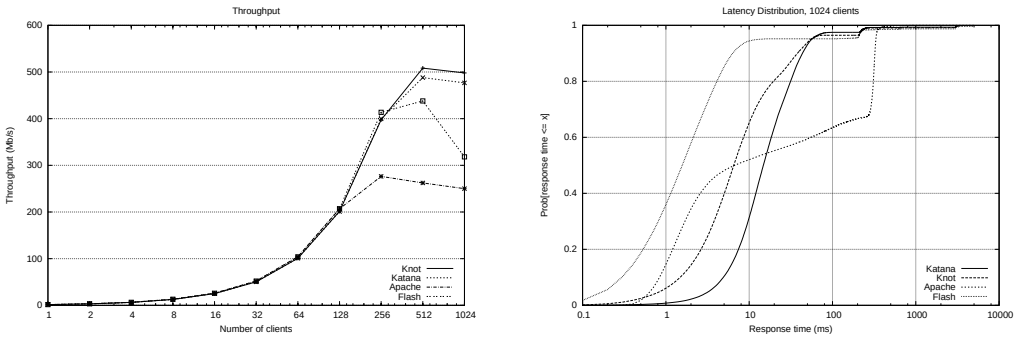

Figure 9: On the left is the throughput of the servers we tested. Each client makes 5 requests every 20 ms for single 4K file. On the right is the cumulative distribution function of request latency for 1024 clients.

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

low median latency, but an even longer tail. Again, the 506 connection limit is probably responsible. Based on the graph, the tail of the Apache distribution is remarkably short, but the median value large. We theorize that Apache accepts connections as quickly as possible, and either handles them immediately or queues them for later.

Based on these results, Katana is as fast or faster than existing static web servers. 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 6.2.2 against the J2EE and .NET results provided by the Middleware Company. The results are shown in Figure 11. 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, despite the fact that the Middleware tests were done using superior hardware. We feel that these results demonstrate the advantages of specializing languages and the runtime to a domain like servers.

7 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 codebases, suboptimal performance, and too much freedom in dangerous areas like concurrency—are the ones that Katana was designed to fix.

Katana's concurrency model draws from the Capriccio thread library [41], 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 to by a library like it. The SEDA framework [42] also addresses the problem of highly concurrent servers, although it focuses on the problem of graceful degradation 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 DSLs is similar to that of many existing languages. The type system and module system draw heavily from OCaml [34, 28], 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 [36],

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 [29], 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 [39], 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 [33], 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 [27] 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 [26] 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 [40, 22, 18, 16, 9, 13]. 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.

8 Conclusion

Katana is a server architecture designed to minimize the cost of developing a web application server. Cost is directly related to the safety and performance of an architecture, and to the productivity of the programmers who use it. Katana optimizes these criteria through the use of domain-specific languages and a specialized runtime. In our analyses and experiments, Katana significantly increases safety, productivity, and performance over existing solutions.

Safety is increased by removing dangerous language features and by statically checking the remaining ones. Katana languages are strongly typed at compile-time. Unlike Java, there are no casting errors, array out of bounds exceptions, or null pointer exceptions. Since shared state is highly restricted in Katana (managed by the runtime), race conditions and deadlocks are impossible.

Regarding *productivity*, Katana supports all the features expected of a modern language—modularity, abstract types, and polymorphism—while providing replacements for the dangerous ones like manual memory management and unrestricted threading, for what we feel is an appropriate level of expressiveness. Additionally, the Katana language framework is extensible. Languages can be modified and new languages created as needed. A shared, extensible compiler infrastructure makes language manipulation easy.

Our specialized languages are concise, declarative, and most importantly, tailored to particular server tasks—a particularly labor-saving combination. In our experiments, Katana programs were shorter than their Perl or J2EE equivalents, in some cases by a significant margin. Development time was low and the time to learn the new languages was also short. Katana's static safety guarantees dramatically reduce the likelihood of bugs, further shortening development time. Built-in session management, a library of web server tools, and native database access also aid developers. Finally, like Java and other web programming languages, Katana is portable across operating systems and architectures.

Katana ensures excellent *performance* via several mechanisms. Since Katana programs are checked at compile-time, there is no need for dynamic checks. Katana's common binary representation eliminates the need for costly marshalling and demarshalling. Memory management is done entirely using regions, exploiting the short-lived nature of server requests. The

Katana runtime is designed around an efficient cooperative user-space threading library for massive scalability. Benchmarks indicate that Katana matches or outperforms existing C- and Java-based static and dynamic web servers.

Based on these results, we see Katana as a promising framework for creating high-performance, expressive, economical, and safe Internet 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.j%sp`.

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

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

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

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

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

[31] Netcraft web server survey. `http://www.netcraft.com/Survey/Reports/0405/byserver/index.html`.

[32] Newsdog. `http://www.ocf.berkeley.edu/~ajs/`.

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

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

[35] 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.

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

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

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

[39] 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.

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

[41] 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.

[42] 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.