Journal of Web Engineering, Vol. 4, No. 4 (2005) 339–371
© Rinton Press

SERVER ENFORCED PROGRAM SAFETY FOR WEB APPLICATION ENGINEERING

HENRY DETMOLD, KATRINA FALKNER, DAVID S. MUNRO & TRAVIS OLDS
School of Computer Science, The University of Adelaide, North Terrace
Adelaide, South Australia 5005, Australia
{henry,katrina,dave,trav}@cs.adelaide.edu.au

RON MORRISON & STUART NORCROSS
School of Computer Science, University of St Andrews, North Haugh
St Andrews, Fife KY16 9SS, Scotland
{ron,stuart}@dcs.st-and.ac.uk

Received January 18, 2005
Revised June 8, 2005

As Web application development evolves from initial *ad hoc* approaches to large scale Web engineering, it is increasingly important to adopt systematic approaches to ensuring safety properties of Web applications. In particular, engineers constructing Web applications should be provided with at least the same guarantees of static safety as in preceding development paradigms; the current absence of such guarantees leads to Web application users being forced to endure failure modes that would never be accepted from conventional applications.

We observe that much is known about program safety in the traditional software development domain. Based on this observation, we contend that Web engineering should adopt an evolutionary rather than revolutionary approach to program safety. That is, existing solutions from conventional development should be evolved to match the exigencies of the Web engineering context, rather than engendering solutions that are wholly new.

With this evolutionary approach in mind, we introduce a categorisation of the problem area into four major safety properties, each related by analogy to a problem in the conventional development paradigm. Further, we observe that in the Web context, these properties are interrelated, and hence adopt an integrated model for their enforcement. Based on this integrated model, we demonstrate an approach to Web application safety that is both simpler and more powerful than previous, non-integrated, approaches. In contrast to previous systems, our approach as implemented in our WebStore application server achieves the safety goals without recourse to new and unfamiliar programming constructs. Finally, WebStone benchmark results comparing our server to existing mainstream Web application development platforms demonstrate that it provides acceptable performance for a wide range of Web applications.

Keywords: Web Applications, Type Safety, Referential Integrity, Persistence

Communicated by: M Gaedke & D Schwabe

1 Introduction

The development of Web applications has evolved from the construction of small applications and gimmicks using *ad hoc* development techniques into the domain of *Web Engineering* [1, 2], which seeks to construct the large scale multi-user applications of the future. One important tool for the conventional software engineer is the notion of static program safety. In a safe programming language, the programming system provides developers with guarantees that program modules that pass a given stage in the development process (typically, compilation) are free from a broad class of defects. As a result, failures deriving from those defects are prevented from occurring during program execution. Whilst this notion of safety has widespread acceptance in conventional development, it is notably absent from the Web engineering field. Consequently, Web application users are forced to endure failure modes that would never be accepted from conventional applications. We categorise important program safety properties of Web applications as follows:

(i) Ensuring all delivered HTML content is syntactically well-formed.
(ii) Ensuring referential integrity of hyper-links in both static and dynamically generated content.
(iii) Ensuring consistency of Web forms with the services processing form input.
(iv) Ensuring statically safe binding of the code of session operations to variables defined with session scope.

Previous work in this area has addressed various of these properties, but has not enumerated the complete set, nor have the inherent relationships between them been recognised. This failure to recognise the relationships significantly complicates the provision of a safety regime.

Violations of these properties result in failures that are exposed to both human and programmatic users of the Web application. From a human user's perspective, these failures may be classified as follows:

- *Malformed pages* – a page served by a Web server is not valid HTML. Consequently, it does not operate as intended with the user's Web browser, perhaps denying the user access to important functionality if part of the page is unable to be rendered.
- *Broken links* – the destination of a link within the application is erroneously deleted or moved, or the application generates a page containing a link that does not refer to an extant object. These errors lead to users experiencing the well known "404 - Not Found" failure.
- *Inconsistencies in links to Web services* – the `ACTION` attribute in an HTML `<FORM>` tag associates the inputs in the form with a Web service responsible for processing those inputs. The form and service can vary independently, but are related by an implicit requirement for consistency which must be maintained. In particular, the form must ensure the submission of all inputs expected by the service and their type correctness. If a service is updated and becomes inconsistent with the forms referencing it (or *vice versa*) then users will experience failures when they submit the form(s).
- *Failed session state linkage* – since session state is not statically scoped, a session-oriented application can fail during execution due to the absence of required state. Many systems mask this failure, by creating the missing session state with default values and with the type implied by the point of use. This type may be inconsistent with the use in other parts of the application. This masking effect leads to erroneous application behaviour, which, whilst less severe in appearance, is in fact more difficult to diagnose and correct than the underlying failure.

Notice that each of these failures results from a violation of the corresponding safety property.

These classes of failure also affect programmatic users of a Web application, such as systems for B2B (business-to-business) commerce, but the impact is potentially more severe. Human users have an inbuilt capacity to tolerate failures, and may be able to devise workarounds enabling them to continue; programmatic users have no such capacity. Known failure modes might of course be handled by special case code, but this programmed failure tolerance comes at the cost of increased program complexity and hence increased development expense.

Our goal is to prevent these failures from arising during the operation of Web applications. We pursue this goal entirely within the confines of standard HTTP. Hence we place no constraints on the external tools and processes Web application developers choose to employ.

We model Web application safety through the integration of the four identified safety properties. The basis of our approach lies in well-known concepts from the programming language domain, in particular, strong typing, higher-order functions, and the preservation of referential integrity. The key advance of our integrated model is simplicity: by addressing all four properties simultaneously, we are able to derive a concise set of interrelated constraints that enforce our safety regime. This integrated model addresses a number of deficiencies in the previous work. First, previous attempts have addressed only a subset of the properties, for example, the W3Objects system [3, 4] addresses link integrity but not the other three properties. Similarly, the `<bigwig>` system [5, 6] addresses several of the other properties, but does not enforce link integrity. Secondly, those previous systems that address several of the properties suffer from increased complexity as a result of considering each in isolation.

The Web Engineering domain is, of course, very broad, and our work is concerned only with part of that domain, at the level of implementation. However, our approach is completely compatible with higher-level aspects of Web Engineering. In fact, we argue that the static safety guarantees that our approach provides improve traceability from preceding high-level activities (such as requirements and design) to implementation, and thereby enhance the degree to which activity at the level of implementation supports the higher-level engineering activity.

Our implementation is based on the representation of the URLs linking Web application components as *typed higher-order functions* [7] in a *persistent system* [8, 9]. The key advance in our implementation is that all content is produced by computation over typed and initialised URLs. Thus, content generators deliver Web services that operate over extant URLs, instead of strings for which system-level guarantees cannot be made. An immediate consequence of this approach is the prevention of broken links within static and dynamically generated content. The primary conceptual contribution of this paper is to show that this approach extends further and provides an integrated Web application safety regime. Within this regime, all four safety properties are statically enforced.

We present the *WebStore* Web application server [10], which provides the consistency guarantees we seek, based on typed referential integrity of URLs, extending across application components. Our system is formulated in terms of HTML for simplicity of explanation and implementation. However, the techniques we describe apply equally to XML. Furthermore, because XML provides both a rich and explicit type system (XML Schema [11]) and a partial model of computation (XML Query [12]) it has the potential to support a more comprehensive program safety regime than is possible with HTML. We note such opportunities as they arise and present a discussion of an XML version of our server in Section 6.

We have measured the performance of our server in comparison to mainstream Web servers. Whilst it would be innapropriate to extrapolate to very large Web service provision, the results presented in this paper do indicate that for at least moderate sized Web services provided on a single server, the safety advantages provided by our server do not result in an unacceptable deterioration in performance.

2 Failures within the Web Service Request Life Cycle

This section presents a more detailed description of the four classes of failure. Figure 1 illustrates the life cycle of an HTTP request for dynamic Web content; each failure is discussed with reference to this diagram.

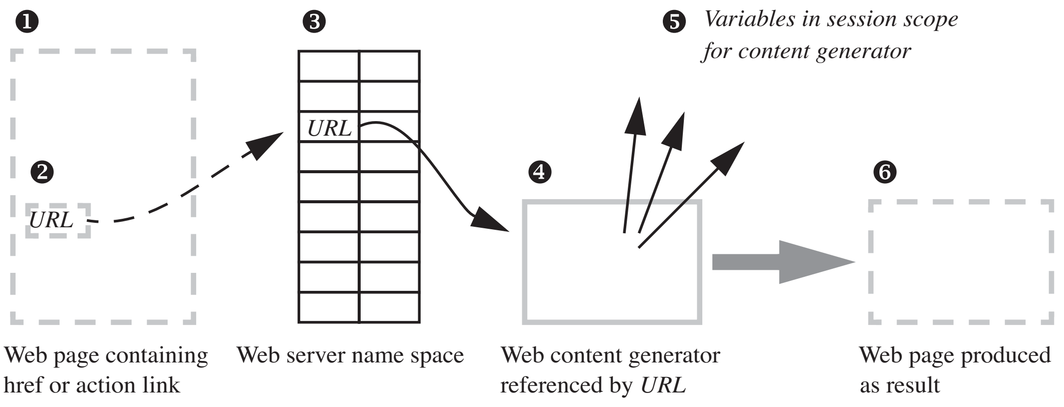
❶
❸
❺ Variables in session scope for content generator
URL
❷
URL
❹
❻
Web page containing href or action link
Web server name space
Web content generator referenced by URL
Web page produced as result

Fig. 1. Web Application Requst Life Cycle.

2.1 Broken links

A Web browser loads a page, ❶, containing a link, ❷. User action causes that link to be resolved, resulting in the browser sending the URL of the link in an HTTP request to the Web server. The Web server attempts to locate the requested URL within its namespace, ❸, but this lookup fails, returning a 404 error message in consequence.

This failure cannot be prevented in the general case, where URLs may be specified in links within remote sites, or even typed in as literal text. However, the vast majority of links within Web applications are internal: from an artifact within a Web application, to another artifact within that same application. In this, the common case, broken link failures should be prevented statically.

2.2 Inconsistencies in links to Web services

Suppose that the URL at ❷ is the `ACTION` attribute of a form containing *name*, *age* and *sex* attributes. Submission of this form will lead to the browser sending an HTTP `GET` or `POST` request to the server. Assume that the processing of the request successfully locates the URL within the name space, ❸, and resolves it to an object (a content generator) in the content space, ❹. However, the content generator turns out to be a function expecting *name*, *age*, *sex* and *email* parameters. The absence of the last parameter prevents the content generator from executing successfully, and a failure results.

The source of this problem is the inconsistency between the object of reference (a function expecting four parameters) and a reference to that object (a form providing only three parameters). Whilst it is of course possible to program defensively to tolerate such situations, it is preferable to avoid the need to do so (and hence avoid complicating the program code) by preventing such inconsistent references from arising in the first place.

2.3 Failures in session state binding

Next, suppose that the request has reached the stage where all necessary parameters are available and the content generator can consequently begin executing. During this execution, the content generator may access non-local variables within its closure. One class of variables that is important in Web applications are those in so called *session scope*, ❺. A *session* is the series of requests from a given user to a given application. Variables with session scope come into existence at some point during a session and continue to exist for the remainder of that session, being accessible in all content generators executed in response to subsequent requests in the session. Shopping carts, which accumulate the items chosen by a user moving through an online store application are a common example of the use of this facility.

In most current approaches to Web session management, there is no single point in the program text where a given session variable is declared and initialised, *i.e.* it is not possible to determine the defining occurrence of a given session variable by inspection of the program source code. Instead, the defining occurrence is within the first content generator in the session that happens to use the variable. This is an inherently dynamic property, established by the order in which an application's URLs are requested within a given session. In adopting this dynamic approach to session variable definition, current systems resemble programming language with dynamic scoping, such as LISP.

Commonly used systems (such as PHP [13]) implicitly define each session variable at the point of first use, with the type implied by the context of use and the default initial value for that type. This can lead to lack of proper initialisation and result in subtle failures in subsequent program operation. These failures are difficult to detect since implicit definition masks their existence. Implicit definition can also cause problems with typing, in that there is no assurance that each use of a given variable implies exactly the same type. In consequence, typing failures can arise dynamically, depending on the order in which a user requests the URLs within a session.

Even with explicit typing and identification of session variables, there is still a problem of inconsistent definition of session state. We present the following example using a Java/C-like syntax in which the ***session*** reserved word indicates a session variable:

```
HTML cartInit(PO p) {
    session Decimal value = 0.0;
    session ItemList items = null;
    // Generate the response page, with
    // links to addItem and checkOut.
    ...
    return new HTML(...);
}
HTML addItem(P1 p) {
    session Decimal value = 0.0;
    session ItemList items = null;
    // Add item to session state.
    value = value + p.cost;
    items = new ItemList(p.name, p.cost, items);
    // Generate the response page.
    ...
    return new HTML(...);
}

HTML checkOut(P2 p) {
    session Decimal value = 0.0;
    session Vector items = new Vector();
    // Sell the items in the cart.
    for (int i = 0; i < items.count(); i++ ) {
        // Sell an item and move on.
        ...
    }
    // Generate the response page.
    ...
    return new HTML(...);
}
```

The problem is simple: the type of the *items* variable in *checkOut* is different to the type used in the other two procedures, and in particular is different to the type in *cartInit*, the procedure intended by the application developer to initialise new sessions. It is likely that *checkOut* implements an earlier design in which the items in the cart are represented by a *Vector* rather than a linked list. Whatever its cause, the inconsistency will result in a failure (assuming *checkOut* is called subsequently to *initCart* within a given session). Different systems will exhibit different failure modes, typically one of the following:

- The session management system associates types with session variables and rejects the attempt to define the *items* session variable with a different type. Consequently, execution of *checkOut* leads to a run-time error, which is presumably reported to the end-user.
- The session management system allows several different variables within a given session to have the same name but different types. Consequently, execution of *checkOut* leads to the instantiation of a new session variable named *items* of type *Vector*, which is initialised to an empty *Vector*. Hence, the loop inside *checkOut* terminates immediately, and no items are actually sold. The effect here is that the session management system masks a failure in the application's function, insulating end-users from reports of that failure, but not of course from its underlying effect.

Notice that in each case the commercial result is the same: customers are unable to purchase anything and the merchant experiences a complete lack of sales!

Traditional approaches to software engineering have well documented solutions to these problems: through a single point of typed variable declaration with mandatory initialisation, static scoping and static type checking. Web applications should be provided with the same levels of safety guarantees.

2.4 Malformed pages

At the end of the request life cycle, the content generator has executed to completion, apparently with success. However, the page it produces, ❻, is not valid HTML (or valid XML, or XML valid with respect to a given schema, *etc.*). This may lead to the occurrence of failures when that page is rendered by a browser, or consumed in some other way that is sensitive to the syntactic validity of the document.

3 A Model for Enforcement of Web Application Safety

The key to a systematic understanding of the failures encountered in the Web application request life cycle is to recognise that they are all related to *typing* and *references*, concepts that are well understood in the programming language community. *A fortiori*, we observe that each of these failures is a manifestation of a problem that is well-known in the programming language domain.

Based on this observation, we present a model ensuring the desired safety properties. This is formulated within the execution model of a program in a strongly typed programming language with automatic storage management (*garbage collection* [14, 15]). At this stage of the discussion, we describe a Web server that is entirely transient, with all data in main memory. This is of course completely unrealistic: Web content must be persistent; we address the requirement for persistence as an orthogonal issue in Section 4.

3.1 Preventing malformed pages

Our model is illustrated in Figure 2, which shows a modified version of the request life cycle from Figure 1, annotated with type information. At ❶ in the figure, the output of the content generator has the type *HTML*, a structured type representing valid HTML content as a Cartesian product type (*view*), as follows:

```
type HTML is view[
    head:  HTMLHead;
    body:  HTMLBody
]
type HTMLHead is ...
```

Type Definition 1

Each object of type *HTML* is in fact the root of a graph of objects representing Web content in structured form. The requirement that all output be valid HTML (rather than a malformed page) implies that the type of each content generator, such as that shown at ❷, must be some function returning *HTML*, *i.e.* *fun(*)→HTML*, with parameter type left unspecified at this point in the discussion.

Now, the Web server name space, shown at ❸, contains a pointer to each content generator. These pointers are as shown at ❹. Furthermore, each namespace entry must be mutable (to

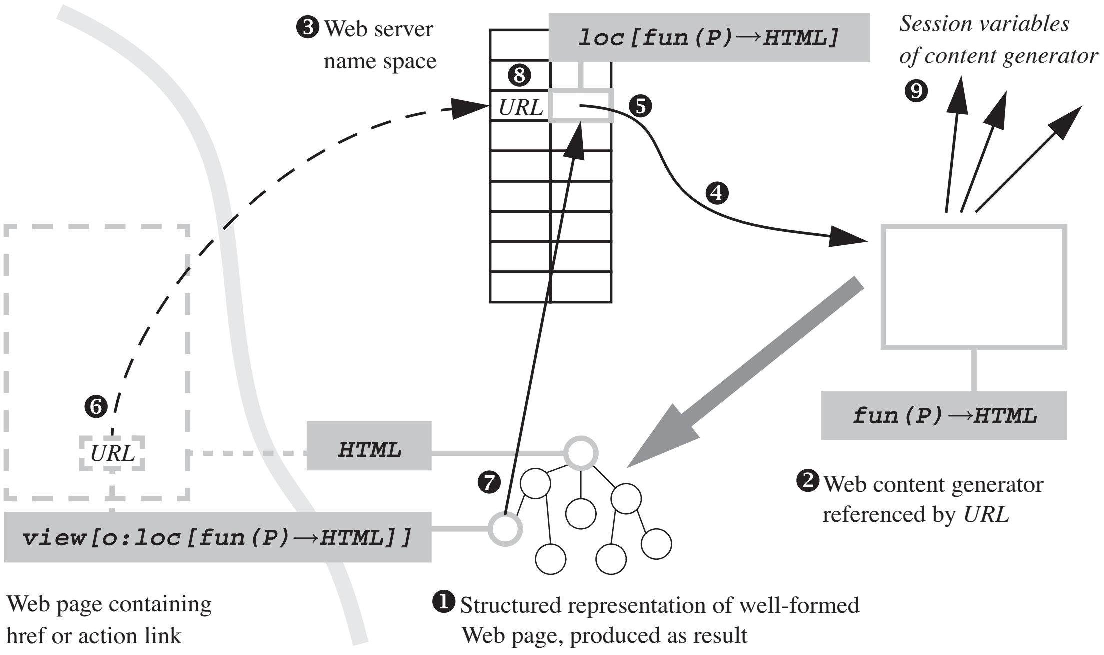
❸ Web server name space
loc[fun(P)→HTML]
Session variables of content generator
❽
URL
❺
❾
❹
fun(P)→HTML
❻
URL
HTML
❼
view[o:loc[fun(P)→HTML]]
❷ Web content generator referenced by URL
Web page containing href or action link
❶ Structured representation of well-formed Web page, produced as result

Fig. 2. Typed Web Application Requst Life Cycle.

allow replacement of content generators by updated versions), so the type of the namespace entry at ❺ is an updateable location (`loc`) holding a value of type *fun(*)→HTML*.

Next, references (via URLs) to this namespace entry, such as ❻, are (implicitly) of some type holding a pointer to a namespace entry. The key point is to model the extra level of indirection from hyper-links to content generators. For simplicity in this discussion, we use a Cartesian product type:

```
type Link is view[
    referent:  loc[fun(*)→HTML]
]
```

Type Definition 2

Given all hyper-links requested within the execution of an application are represented by objects of type *Link*, it follows that all such requests will resolve to functions returning values of type *HTML*. These structured values are then converted to HTML text by an *unparsing process*. Finally, given that this unparsing process produces correct HTML text for any *HTML* object graph, the application will generate only valid HTML output - *i.e.* that the *malformed pages* class of failures is statically prevented. Hence, we achieve the first safety goal: ensuring that all pages generated in our system are well-formed HTML.

3.2 Preventing broken links

Notice the inherently circular nature of Figure 2: at the beginning and end of every Web request there is a page with structured representation of type *HTML*. Now, given that referential integrity is maintained at the language level, for both the pointer, at ❼, representing the link, and for the pointer at ❹, and that the unparsing process when applied to the pointer, ❼, produces the URL, ❽, then it follows that no 404 error will result from the request for that URL. Furthermore, it follows by induction from a given request's life cycle to the next, that no 404 failures will be experienced when following links within our system, *i.e.* the *broken links* class of failures is prevented for all links internal to the site, in both static and dynamic content. Hence, we achieve the second safety goal: ensuring referential integrity of links in both static and dynamically generated content.

This point requires further elaboration: links are always represented as typed entities, (of type *Link*, in fact), both for storage and when used in server side computations. In particular, a server side computation that wishes to output a link must output an initialised value of this type; a value that is guaranteed by the referential integrity rules to refer to a valid referent object. The unparsing process must render these *Link* objects as text for interpretation by browsers, but this process operates entirely at the system level, is not in any way exposed at the user level, and generates only valid links.

For most content generators, *Link* objects are inserted into the namespace during the process of constructing an application, and are given an URL at that time. However, it is possible for the execution of a content generator to create new *Link* objects, and omit to insert them into the namespace. To ensure that this does not lead to broken link failures exposed to users, the unparsing process assigns such links unique machine generated URLs and inserts them into the namespace.

3.3 Preventing inconsistencies in links to web services

A content generator implementing a Web service, such as that shown at ❷ in Figure 2, is a function from a set of name-value pairs to a result of type *HTML*. The natural way to model these name-value pairs is using a Cartesian product. Hence, a content generator is a value of type *fun(P)→HTML*, where *P* is some Cartesian product type containing a field for each of the name-value pairs[a]. This implies that the namespace entry is an updateable location holding a value of this type (*i.e.* namespace entries are values of type *loc[fun(P)→HTML]*), and in turn that links to the content generator are objects containing references to *loc[fun(P)→HTML]* values. Such links typically occur as `ACTION` attributes within Web forms. For a form to refer to a content generator, the form must provide inputs corresponding to each and every field of *P*.

The Web page type, *HTML*, is a complex structure. It includes *loc[fun(P)→HTML]* elements as leaf nodes, in order to model links (including form `ACTION` attributes), contained within Web pages. However, the parameter type, *P*, may be different for different links within a page. For example, `ACTION` attributes within distinct forms may well refer to content generators expecting different parameters, *i.e.* different types for *P*.

The constraints in the model of consistency of links to Web service are as follows:

(i) The model must enforce consistency of inputs between a form and the content generator linked via that form's `ACTION` attribute. *I.e.* a constraint is needed to ensure that when forms are constructed within the WebStore, or introduced to it via upload, then each form is consistent with the content generator to which it is linked.

[a]A degenerate case is a content generator taking no parameters, in which case *P* is the empty Cartesian product. Static content, amongst other things, is modelled this way.

(ii) In order for it to be possible to define the *HTML* type in finite form, a single type, *Link*, is needed as the type of all links. *I.e.*, from the point of view of *HTML* structures, all *Link* objects appear to have the same abstract type.

(iii) However, the model must ensure that when a link is constructed, it has an actual type that encodes the inputs expected by the content generator. *I.e.* from the point of view of *Link* object creation, *Link* objects may have an actual type differing in terms of the formal parameter type, *P*, which must be some sort of view type.

(iv) Furthermore, this variation (in the type *P*) must be the only variation between the actual types of different links. *I.e.* all links must refer to functions returning *HTML*. This constraint is necessary to ensure that content generators produce valid HTML.

(v) Finally, it should be possible to update *Link* objects, without compromising the static guarantees of input consistency. *I.e.* links may be updated so as to refer to some other content generator, but, to ensure consistency of expected inputs, the updated value must be of the exact same actual type as the original.

There are two cases where constraint (i) operates: that of forms uploaded to the WebStore, and that of forms generated as a result of computation within the WebStore.

When forms are uploaded to the server, a check is made that the form text is consistent with the type *P* (for the content generator linked to the form action). The upload is rejected if there is an inconsistency. That is, there is a dynamic check, but this does not compromise our claim to static safety, since it occurs before the page can possibly be requested. Uploaded forms are the most common case, since developers typically desire the ability to edit forms off-line.

Ensuring constraint (i) is more complex for forms that are dynamically created in response to HTTP requests. In these cases a dynamic failure is not permissible, since the request is already in progress. Therefore, the model assumes that the type *P* is authoritative, and requires the unparsing process to insert `<INPUT>` tags into the form so as to make the form consistent. The effect is that the system statically guarantees that a dynamically generated form is usable. The only drawback is a possible degradation in its visual presentation.

To satisfy constraints (ii) to (iv), we need a type constructor that allows the definition of a single abstract type, *Link*, but permits variation in the formal parameter type *P*. Furthermore, we need to support assignments that update *Link* objects so that they refer to different content generators, without voiding static guarantees of consistency between forms and content generators. The *Link* type satisfying constraint (ii) is a Cartesian product, as follows:

```
type Link is view[
    referent:  any[loc[fun(P < view)→HTML]]
]
```

Type Definition 3

In this type, the field, *referent*, is a *constrained infinite union.* The constraint given within the type of the field is that it must contain a value of some type that is a location containing a function from *P* (constrained to be a view type) returning *HTML*.

When a given *Link* object is constructed, a location with a specific actual type *fun(P1)→ HTML*, where *P1* is some sort of view (*i.e. P1* is an instance of *P*), is created and injected into the infinite union. *I.e.* constraint (iii) is satisfied. The type checker statically ensures that the value injected is indeed a location containing a function returning *HTML*. *I.e.* constraint (ii) is satisfied.

In order to access the value in the field, either to use the value within the location, or to assign to it, a dynamic type check is required. This is performed by a language level projection operation such as in the following code:

```
let updateLinkDestination :=
fun(lnk: Link, cg: fun(P1)→HTML)
begin
    project lnk.referent onto
        type loc[fun(P1)→HTML]:
            lnk.referent := cg;
end
```

This example code updates the *referent* field of a link to refer to some other content generator. The update succeeds providing the field's actual type is *fun(P1)→HTML*, for some particular type *P1*. If this is not the case then this code does nothing. To generalise, any update of the link can only replace the value within the typed location with another of the same actual type. *I.e.* constraint (iv) is satisfied.

It is also necessary to ensure that the referent object of a link is some function returning *HTML*, otherwise the guarantee of well-formed HTML is not maintained. The statically checked constraint on injection into the constrained any (the *referent* field of *Link*) enforces this property.

HTML is rather limiting of the extent to which forms can be checked for consistency with related services. In contrast, XML XForms [16], allows detailed specification of the types of input fields, thereby enabling our approach to perform comprehensive consistency checking, of input types as well as names.

3.4 Preventing failures in session state binding

The various failures observed in session state binding arise because Web development has deviated from the design decisions taken by mainstream programming languages in relation to issues like scope, typing and the requirement for a single point of variable initialisation. The decision to abandon static scoping is the root cause of this problem, and consequently we reject it.

The crux of our solution is that the content generators providing operations on session state are higher-order function closures that are dynamically generated during the session. Typically, these dynamic content generators are by-products of the execution of the content generator that began a session. Furthermore, they are lexically nested within the scope of that first content generator and consequently have statically scoped access to its local variables, ❾ in Figure 2. These local variables provide the session state.

We illustrate our solution using a small, shopping cart style application, similar to that given previously in Section 2.3. The code of the first dynamic content generator in each session is:

```
let cartInit := fun(p: P0) → HTML
begin
    ! Session state.
    let value := 0.0
    let items := nil(ItemList)
    let addItem := fun(p: P1) → HTML
    begin
        ! Add item to session state.
        value := value + p.cost
        items := new ItemList(p.name, p.cost, items)
        ! Generate the response page.
        ...
        return new HTML(...)
    end
    let checkOut := fun(p: P2) → HTML
    begin
        ! Sell the items in the cart.
        let curs := items
        while (curs != nil(ItemList))
        begin
            ! Sell an item and move on.
            ...
        end
        ! Generate the response page.
        ...
        return new HTML(...)
    end

    ! Generate the response page, with
    ! references to addItem and checkOut.
    ...
    return new HTML(...)
end
```

There are two session state variables in this application, *value* and *items*, shown underlined wherever they occur in the source code. These are local variables: separate instances are created during each call to the *cartInit* function, *i.e.* each call to *cartInit* starts a new session. Notice that the *addItem* and *checkOut* functions refer to the session state: this access is both statically scoped and statically typed. Notice also that the problem of consistent and proper initialisation is addressed: each session variable is initialised at the (single) point of declaration.

Now, the page generated by a given call to the *cartInit* function includes references to the *addItem* and *checkOut* functions. These references represent hyper-links within the application. When the user clicks on such a link (to *addItem*, say), the Web request life cycle occurs as before, and the appropriate function (*addItem*) is run to generate the next page. This relies on a block retention stack model, such that the stack frame of a given activation of *cartInit* remains in existence whilst there are reachable references to functions within that activation (such as links to *addItem* and *checkOut*). The key requirement is the storage of closures, not higher-order functions *per se*: in Java we could use inner classes to the same effect.

The links to *addItem* and *checkOut* are examples of a problem that we have seen previously, namely ***Link*** objects that are not recorded within the server namespace. Given that this problem is solved as before, then the model prevents the *failed session state linkage* class of failures and achieves the fourth safety goal.

3.5 Summary of the model

The constraints the model imposes to ensure the four static safety properties are:

(i) All updates of Web content occur via assignments that are subject to strong type checking.
(ii) Referential integrity is maintained for all pointers.
(iii) All static and dynamically generated pages are represented in structured form by a graph of objects rooted at an object of type ***HTML***.
(iv) All links within pages are represented by objects of the constrained infinite union type ***Link***, given in Type Definition 3 on page 348. Moreover:

    (a) Content generators referenced by these links are higher-order functions returning values of type ***HTML***.
    (b) The operations of session oriented applications are higher-order functions lexically nested within some content generator and accessing local variables of the enclosing content generator, via conventional static scoping. These local variables provide the session state.

(v) A system level unparsing process is employed to render structured page representations into HTML text. This ensures that any ***Link*** object it encounters is represented within the Web server namespace, inserting a given link with a machine generated URL if necessary.
(vi) A system level upload and parse process ensures that forms within static pages uploaded to the server are consistent with the content generators to which they refer.
(vii) The system level unparsing process ensures that dynamically generated forms it encounters are consistent with the content generators to which they refer, by adding `<INPUT>` tags where necessary.
(viii) There is an underlying requirement that the content generators, pages and links in the model persist across server shutdowns and crashes, whilst maintaining all of the above constraints.

These constraints form the design specification of our implementation of the model, which is made manifest in the *WebStore* Web application server.

4 WebStore Implementation

Two key requirements have guided our approach in implementing a research prototype version of the WebStore. These are:

- The program safety model is based on several notions from strongly typed programming languages with safe pointers. In particular, content and content generators are represented by programming language objects and hyper-links are represented by typed, safe pointers. Consequently, our prototype implementation requires the support of a strongly typed programming language with safe pointers.
- Web content and content generators are long-lived entities. They need to survive both controlled shut-downs of the WebStore and server hardware failures. That is, these entities have a requirement for persistence.

Taking these two requirements together, we have chosen to implement our prototype in our ProcessBase system [17], which provides a strongly typed programming language with safe pointers and supporting orthogonal persistence.

4.1 Orthogonal persistence

The implementation of the WebStore is based on a pre-existing and proven programming technology known as *orthogonal persistence* [8, 9]. Orthogonally persistent systems such as ProcessBase improve programmer productivity by providing a uniform model for operating over all data, regardless of whether that data are short-lived (transient) or long-lived (persistent). Specifically, persistent programming systems provide the standard programming language operations such as assignment, expression evaluation, dereference and function call as the means to operate on all data. The WebStore has unusually intensive and complex requirements for the manipulation of persistent data, and the observed benefit to programmer productivity has been commensurately large.

Orthogonal persistence avoids both the development cost and risk inherent in *ad hoc* arrangements for data translation and long term storage. Instead, these facilities are provided entirely at the system level. In systems based on persistence by reachability, such as ProcessBase, the objects that persist are those reachable in the transitive closure of one or more root objects. In particular, there is no requirement for programmers to identify persistent objects. This property of orthogonal persistence has been of great benefit to the work described in this paper, in that it has enabled us to avoid explicit consideration of persistence within the safety model, thereby simplifying the model and enabling concentration on its fundamental properties.

In the same way as there is a uniform model for operation over all data, orthogonal persistence provides a uniform model for data protection. All data are protected against improper use in the same way, again regardless of lifetime. Specifically, in persistent programming languages such as ProcessBase, data are protected by a programming language type system. The WebStore safety model is based around programming language type system concepts, an approach that is only possible with the uniform protection model provided by orthogonal persistence.

Persistent systems (whether orthogonal or not) are inherently long-lived and must therefore support system evolution, preferably in an incremental and non-disruptive fashion. The systemic facilities for incremental system evolution provided in ProcessBase have proven highly beneficial in the construction of the WebStore, where system change is continuous, through the addition and update of content generators within the Web site.

Finally, orthogonally persistent systems preserve referential integrity over the entire computational environment for the whole life-time of a persistent application. This includes the preservation of the integrity of references from code to non-code (*i.e.* ordinary) data and *vice-versa*. This point in particular convinced us to base our prototype WebStore on an integrated persistent system, rather than choosing to manage data in an external database.

4.2 The ProcessBase system

ProcessBase is the latest in a long series of research prototype persistent programming languages, with predecessors including PS-Algol [18] and Napier88 [19]. The system consists of two main parts: a byte-code interpreter, incorporating a memory cache, and a persistent object store, managing secondary storage. The interpreter is similar to early Java implementations, in that no effort has been expended on complex optimisations for just in time compilation to native code.

The primary distinguishing feature of ProcessBase is its type system. Salient points relevant to this paper include:

- Higher-order functions as first class data values, able to persist, to be passed as parameters and returned as results and to be assigned to variables.
- A distinction between mutable and immutable data, enforced at the type system level.
- The provision of an infinite union (***any***) type, supporting a dynamically type checked operation for projection of a previously injected value.

The model presented in Section 3 uses these type system features extensively. However, there are similar features in most modern strongly typed programming languages (for example, inner classes in Java are an alternative to higher-order functions for supporting the session state model). Hence instantiation of the model is language specific but the conceptual basis is more generally applicable.

4.3 Operation of the WebStore

Given the facilities provided by ProcessBase, implementation of the WebStore model is relatively straightforward; the principal complexities lie in the formulation of the model. The important processes in the operation of the WebStore are:

- Publication of Web artifacts.
- Serving HTTP requests.
- Reclamation of storage allocated to obsolete Web artifacts.

Figure 3 shows the operational structure of the WebStore at a high level. The diagram shows how content generators, including static pages, are stored in structured form. Notice the complete correspondence between URLs in the Web space and pointers in the structured object space. A system level data structure, the URL-to-object map is used to translate between URLs and pointers as data move in either direction across the boundary between the two spaces.

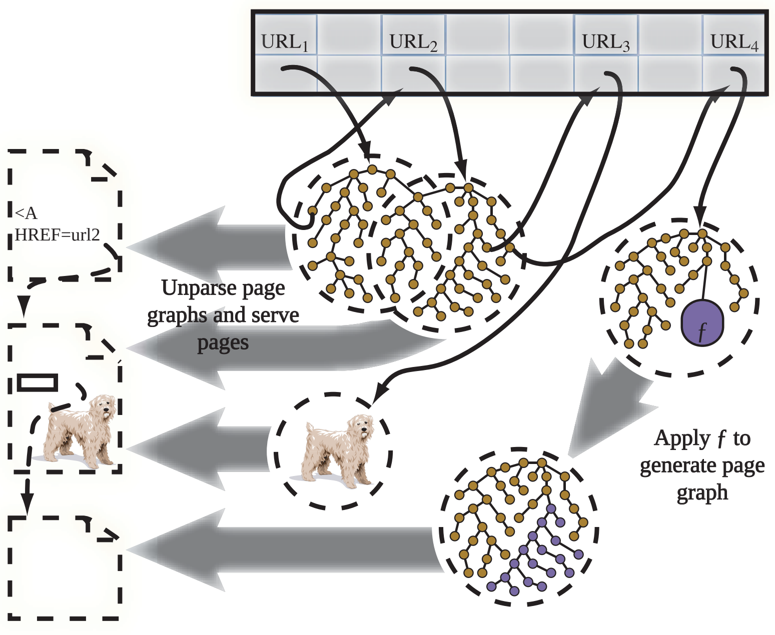
URL$_1$
URL$_2$
URL$_3$
URL$_4$
<A HREF=url2
Unparse page graphs and serve pages
Apply *f* to generate page graph
f

Fig. 3. WebStore Operation: Storage Structure and *Unparse-and-serve* Process.

4.3.1 Publication

Requests for publication include both an entity to be published and the URL under which it is to be published. The entity is either a static page or the source code of a content generator.

As static pages are uploaded for publication, they are parsed into a structured form as described in the safety model. The interesting part of this parsing is the handling of URLs:

(i) The URL is checked to see if the host part refers to the Web site served by the WebStore. If not, the URL is stored as a string within the parsed result and the remaining steps are skipped.

(ii) The URL-to-object Map is searched to find a *U2OMapCell* object for the URL, shown at ❶ in Figure 4. If no object is found, the URL is a broken link, and the attempt to publish the page is rejected.

(iii) From the *U2OMapCell* object, the process moves to the corresponding *U2ORef* object, ❷. It uses type introspection on the *updateable* field, ❸, to determine the actual type of the linked object, and executes a projection operation to obtain the value at ❹. This projected value is used to construct a *Link* object, ❺, which represents the URL in parsed form.

(iv) For links to content generators expecting input parameters, which typically occur within HTML forms, there is an additional check, described in the model at Section 3.3. This rejects the attempt to publish the page if the inputs in the form are inconsistent with those expected by the content generator.

Dynamic content generators are uploaded as source code. Once uploaded, these are compiled and published as executable functions.

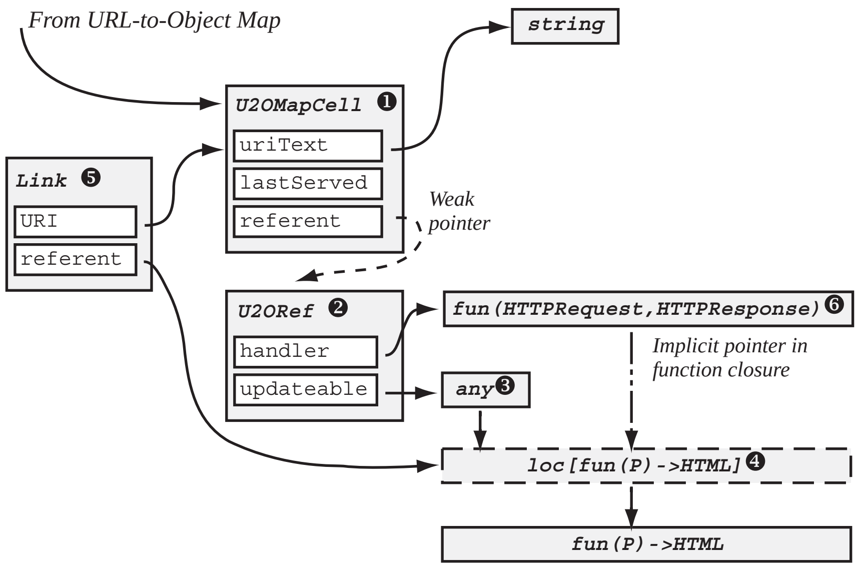

Fig. 4. Details of Link Structure.

The next step in either case is for the publication process to determine the type of the uploaded entity. The type of static pages is *fun(view[])→HTML*. Dynamic content generators are of type *fun(P)→HTML*, for some view type *P*. The exact type *P* is determined during compilation of the content generator.

Next, the URL-to-object map is searched to determine whether the URL provided in the publication request is already served by the WebStore. If so, then the publication process projects the value of the *updateable* field in the *U2ORef* object of the *U2OMapCell* object found in the search. The projected value is an updateable location of some type. If this type matches the type of the uploaded entity, then the entity is assigned to the location and publication completes, otherwise the attempt to publish is rejected. If the URL in the publication request is not found in the URL-to-object map, then a new *U2OMapCell* object is created along with the various objects it refers to (including the published entity).

Notice, at ❻, the *handler* field of the *U2ORef* object. This function serves a role similar to an RPC skeleton, it provides an interface to all Web artifacts that is of a uniform type. This function is called during the process of serving a given artifact, avoiding the need for a dynamic type projection in that (common) operation. The publication process uses *linguistic reflection* [20], to generate an appropriate handler the first time a given URL is used.

4.3.2 Serving HTTP requests

When an HTTP request arrives, the URL-to-object map is searched for the requested URL. The ***handler*** field of the ***U2ORef*** object referenced from the ***U2OMapCell*** object found in the search is called, and performs the following:

(i) Unmarshalling input parameters passed in the HTTP request, into a Cartesian product of appropriate type.
(ii) Calling the function stored in the location associated with the handler, passing the input parameter object in the call.
(iii) Unparsing the result produced by that function, and sends the unparsed text back through the HTTP socket.

The unparsing step has been described within the model. Specifically, the unparsing process uses the ***URI*** field of each ***Link*** object to get the corresponding ***U2OMapCell*** object and then uses the ***uriText*** field of that object in the unparsed link text.

4.3.3 Reclamation of storage

Storage allocated to Web artifacts is reclaimed by garbage collection of the persistent store: there is no explicit deletion. Conceptually the garbage collector operates by tracing from the root object of the persistent store (the root of persistence). Objects that are not reachable in the transitive closure of the root object are identified as garbage and the storage allocated to them is reclaimed.

At the level of Web artifacts, there are one or more distinguished root Web pages, the representations of which are always reachable from the root of persistence. Conceptually, in the Web space, the set of pages retained in the Web server is that reachable in the transitive closure of the root pages. Within the object space, this corresponds to the set of page representations reachable from the representations of the root pages. Hence, the storage allocated to the representation of a page is de-allocated only when that page is not in the transitive closure of the root pages, and there is therefore no possibility that the page might be requested via dereferencng a link within the content served by the WebStore.

However, there are two problems. First, as illustrated in Figure 3, all Web artifact representations are reachable from the URL-to-object map. Since this map is reachable from the root of persistence, the effect will be that Web artifacts never become unreachable and hence never become garbage[b]. The solution to this problem is to distinguish pointers from the URL-to-Object map to the roots of Web artifact representations as *weak pointers* (also known as *weak references*). The garbage collector ignores such pointers when tracing. During the collector's reclamation process, weak pointers that refer to objects that are de-allocated are set to the distinguished *null pointer* value. A reaper process periodically scans the URL-to-object map and removes entries containing these null pointers.

The second problem is related to the timing of successive requests. It is illustrated by the following sequence of events:

- A page (A) containing a link to a second page (B) is served to a Web browser.

[b]Note, however, that the objects that represent a given version of an artifact can become garbage after a new version of that artifact is published.

- Page A is modified so as to remove the link to B, and since the only link to B is within A, B consequently becomes unreachable, and is de-allocated.
- The Web browser still has a copy of A loaded and the user then clicks on the link to B, resulting in a broken link.

Our solution to this problem is as follows:

- All artifacts are served with an expiry date not more than some number of days into the future (given by the system constant *lagDays*).
- When an artifact is published, the pointer from the URL-to-Object map for that artifact is initially a strong (*i.e.* normal) pointer and is reset to a strong pointer whenever the artifact is served.
- Each URL has a *lastServed* attribute associated with it. This is set to the current time whenever the artifact is served or a new version of the artifact is published.
- A process periodically scans the URL-to-Object map, looking for URLs having *last-Served* values more than *lagDays* days in the past. The pointers associated with such URLs are rendered weak.

The overall effect is that the operation of the garbage collector in respect of objects reachable through the URL-to-object table lags sufficiently behind artifact expiry times to ensure that a browser honouring expiry times does not display broken links. Note that the browser would need to adopt a slightly stronger interpretation of the notion of expiry time: not only should an expired page be re-fetched from the server, but any display of such a page should be either refreshed or (if the page no longer exists) removed once the expiry time has arrived. However, this stronger definition of expiry does not seem unreasonable.

5 Performance

In order to gain an understanding of the performance characteristics of our prototype implementation, we have conducted measurements for a number of different content types and for various numbers of concurrent clients. In order to provide a basis for comparison, these measurements have been repeated with mainstream Web servers. It would be inappropriate to extrapolate from these results to make a comparison with industrial strength application servers, and we have not done any direct comparison with such systems. Instead, we focus on the application of WebStore as an alternative for Web applications hosted directly on Web-servers, an important sub-class of Web applications overall.

5.1 Experiments

The degrees of freedom in our experiment are the content type, and the number of concurrent clients. The content types include:

- *Static* – A six kilobyte static page.
- *SSI* – A page including a standard header and footer, through the use of a server side includes mechanism.
- *Query* – A dynamic content generator querying a database of people and organisations, for information about a given organisation and the people related to it.
- *Report* – A dynamic content generator producing a report on all people in the database matching a simple predicate.

Experiments have been conducted for various numbers of clients, between one and a hundred. The test server is a 1.2GHz Athlon uni-processor with 512M RAM, running GNU/Linux.

We use a version of the *WebStone* benchmark tool [21], rather than the more widely known SPECweb99. The reason for this is the SPECweb is intrinsically tied to the measurement of conventional Web server architectures, in that it prescribes aspects of the server architecture (executing Perl scripts via CGI *etc.*). The WebStore architecture is fundamentally at odds with such an approach. In contrast, WebStone is a generic benchmark tool, capable of testing any user-supplied workload, rather than a specific benchmark with a prescribed workload. This is an advantage in that it avoids prescribing aspects of server implementation (and hence can be used with WebStore) but is also a disadvantage in that there is no generally accepted workload for use with WebStone.

5.2 Comparates

For the static content and server-side includes cases we compare the WebStore against Apache (version 1.3.23), the W3C's Java-based server, Jigsaw (version 2.2.1 on Java 1.4) and a simple content management system (CMS) storing content in a relational database. Apache is chosen as it is a de facto standard. Jigsaw is chosen since it is implemented in a language based on a virtual machine, an implementation strategy comparable to that used for ProcessBase. Finally, we test against a CMS built using a relational database system as this is a *de facto* standard in current practice for Web content management.

For the dynamic content tests, we use two comparates, namely PHP within Apache (another *de facto* standard approach) and Java Servlets running within Jigsaw. In both cases, the comparate applications access a Postgres relational database server (version 7.2.1). The database has been indexed to support the queries made by the application. We do not repeat the Servlet tests with Apache, since the Apache project Servlet support (Tomcat) is essentially the same as that in Jigsaw.

5.3 Results

Using version 2.5 of *WebStone* [21], we measured the performance of the Webstore and the several comparates, for each content type and for each number of concurrent clients. We used *WebStone* to gather results over a two minute period in each case, and we repeated each experiment five times.

For static content, the primary concern is throughput, since client caching is effective in reducing latency. Figure 5 gives throughput for the WebStore, Apache and Jigsaw, as measured by connection rate. The WebStore outperforms both comparates for this test case.

For server side includes (SSI), throughput is arguably the primary concern, since the content changes relatively infrequently and client caching is therefore effective in reducing latency. Figure 6 gives the connection rate for the three servers. Notice that the WebStore results for this content type are only slightly worse than for static content, whereas for the other servers there is a marked degradation. This is easily explained: in other servers, each SSI request requires the server to parse one or more files, whereas in the WebStore, content is stored in parsed form.

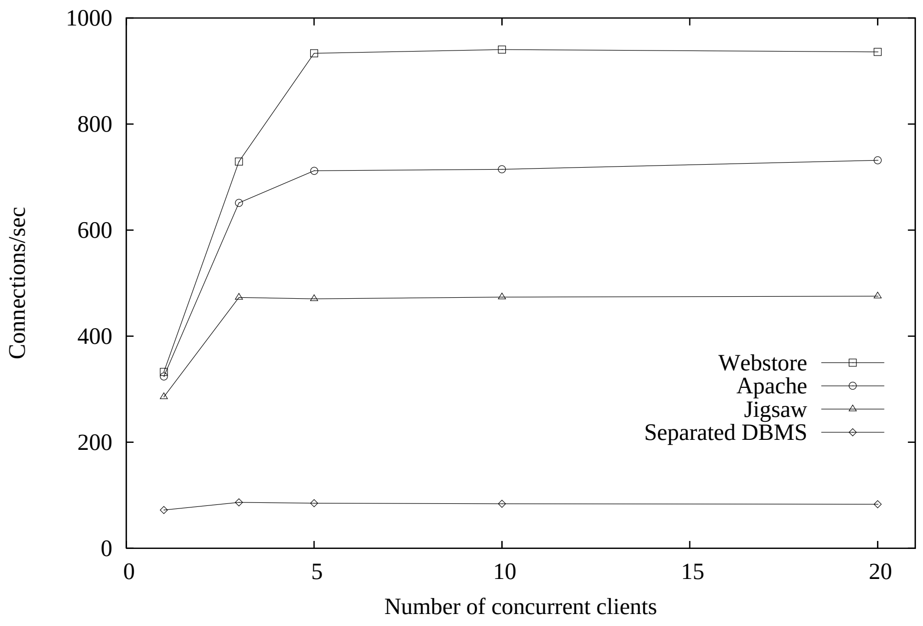

Fig. 5. Connection Rate for Static Pages.

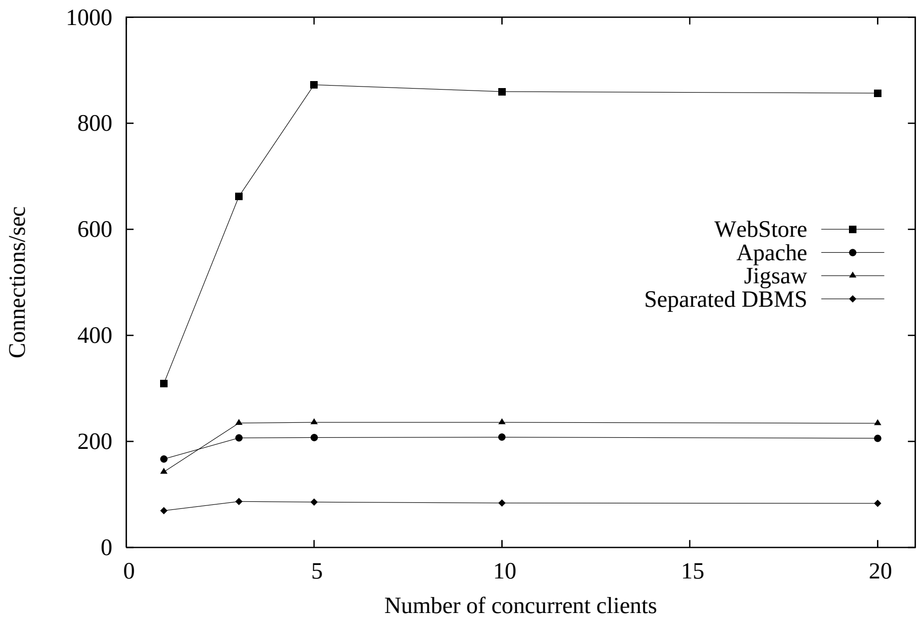

Fig. 6. Connection Rate for Server Side Includes.

In contrast to the first two content types, dynamic content is inherently un-cacheable, and so the latency of requests to the server becomes important. This is the primary determinant of application responsiveness, which is a significant contributor to application usability. Users perceive increased latency in a step like fashion, with precipitous reductions in users' perceptions as latency exceeds certain threshold durations. Established results [22, 23] place these thresholds as follows:

- **0.1 second** – latency threshold at which the user notices a visual disruption in the operation of the application.
- **One second** – threshold at which a typical user's concentration begins to diminish.
- **Ten seconds** – threshold at which a typical user will "turn off" and transfer attention from the application to other work.

From this perspective, application latency can be considered a "soft real time" requirement, with the thresholds being soft real-time deadlines.

Figure 7 shows mean response times for WebStore and Jigsaw, for the two dynamic content test cases. Both systems degrade linearly with increased client count, indicating significant scalability. WebStore is once again significantly better, and in particular manages to keep the mean latency below the first threshold, even with twenty concurrent clients.

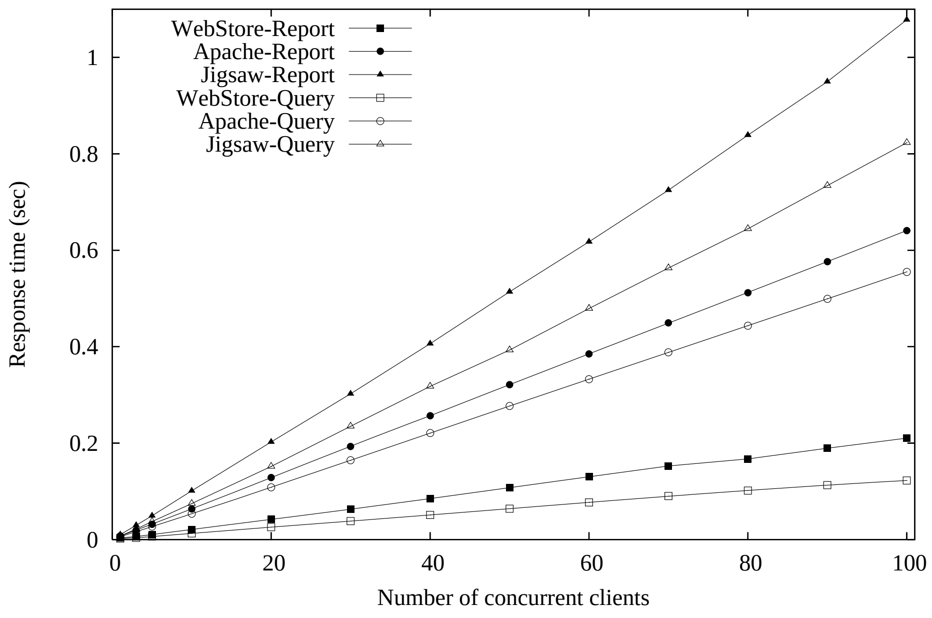

Fig. 7. Mean Response Time for Dynamic Content.

It should also be noted that a significant amount of additional application programming effort was required to improve the performance of the Jigsaw Servlets to the level reported in the graph. Prior to devoting effort to connection pooling and other enhancements, the Jigsaw Servlets were about three times slower than as shown. In contrast, no application level programming effort beyond the implementation of core functionality was required for the WebStore applications.

Mean response time may not be the best metric. If we consider the system as soft real-time, then a more appropriate measure would be the percentage of requests that are successfully completed before each of the three deadlines. The data presented in Figure 8 and Figure 9 evaluates performance of the two applications in meeting the 0.1 and one second deadlines. In the case of the ten second deadline, all systems being compared completed all requests within this deadline.

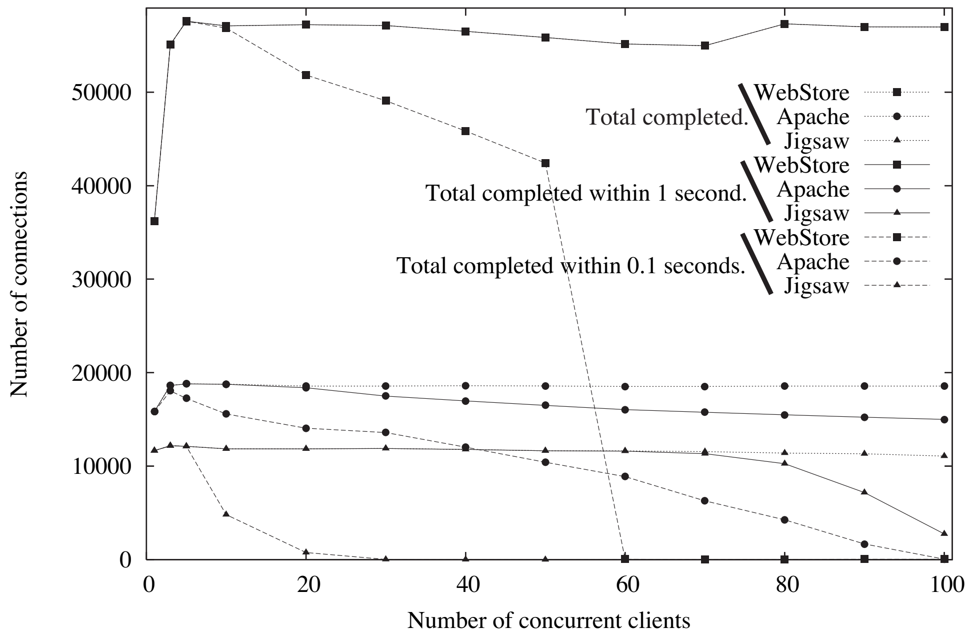

Fig. 8. Response Interactivity for Filtered Report Application.

In relation to the 0.1 second deadline, the applications in the comparate systems begin to degrade with a relatively small number of concurrent clients (ten or less). This degradation is manifest in user-perceptible delays - *i.e.* users will perceive the application as being slow to service some requests at this point. At fifty or more users, very few requests in the comparate systems meet the 0.1 second deadline.

In contrast to the comparate systems, Webstore reaches fifty users with little or no degradation, and then performance against the 0.1 interactivity metric deteriorate precipitously. It is reasonable to regard the WebStore (on the hardware for the experiment) as having reached its capacity at around the 50 client mark.

In relation to the one second deadline, a similar effect (gradual degradation) can be observed for the comparate systems. For the WebStore, even with 100 clients, there is no significant failure to meet the one second deadline, but there is every reason to believe that a precipitous degradation will occur at some larger number of clients, as it does at 50 clients for the 0.1 second deadline.

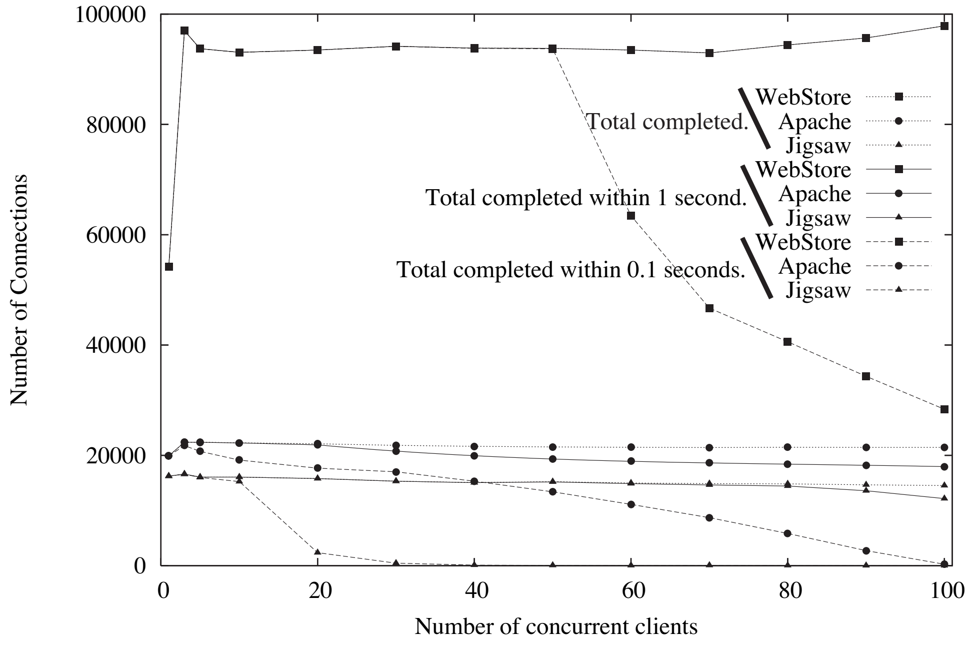

Fig. 9. Response Interactivity for Single Query Application.

There are two main observations to be made. First, that WebStore provides superior interactivity up to a point at which the capacity of the hardware to support it is reached. Second, once capacity is reached, it may be preferable to undergo graceful degradation as in the comparate systems, rather than an adverse phase change as in WebStore. It would seem reasonable to expect that an application specific approach to scheduling the processing of requests could achieve this hybrid, and this is the subject of current investigation [24].

Our experiment is in a sense a stress test for servers, since the clients are single threads that transmit requests as rapidly as they are able within the constraints of HTTP (*i.e.* there is no delay between reception of a reply and transmission of the next request). This is somewhat unrealistic, since in reality, there would be a delay between reception of a reply and transmission of the next request, during which the user decides what to do next. This delay could be modelled by a suitable random process. We do not expect this to alter the character of the result significantly, although it would obviously have an effect on the points at which phase changes occur and possibly on the slope of degradation.

5.4 Summary and analysis

To summarise, the results presented above demonstrate that the WebStore achieves performance comparable to mainstream uni-processor Web servers for static content and performance for dynamic content that is superior by an order of magnitude in comparison to such servers. As a result of the restriction to uni-processor servers, the WebStore performance results should be seen primarily as indicative that WebStore has clear potential to deliver acceptable performance whilst offering improved safety, not that it delivers improved performance in addition to improved safety, when deployed in large scale commercial environments. Specifically, large scale deployments involve server farms consisting in large numbers of (sometimes two-way or four-way multi-processor) servers. Whilst we have no reason to believe that multi-processor implementation of WebStore raises fundamental challenge, efficient distribution of WebStore across a server farm is another matter entirely. Since the use of server farms is the primary means of obtaining acceptable performance for large sites, this is a significant limitation for WebStore.

The above limitations on the applicability of performance results being noted, it is nevertheless meaningful to identify ways in which the WebStore structure acts to enhance performance in relation to dynamic content. The primary factors contributing to the superior performance of WebStore for dynamic content prior to the point at which the system becomes saturated are:

- The static (publication time) approach used in the WebStore for enforcing safety properties minimises the need for extensive (and expensive) checking at request time.
- The integrated approach to data management provided by the persistent system within which the WebStore is implemented eliminates the overheads associated with context switching and connection management that are inherent in any approach using an external database management system.
- In relation to server side includes, and to shared content more generally, the graph based nature of content storage in the WebStore removes the penalty associated with the use of shared content in traditional approaches. As a consequence, performance of the WebStore for this class of dynamic content is indistinguishable from performance for static content.

As mentioned previously, the performance results cannot be extrapolated to industrial strength application server architectures involving load balancing across large server farms. Nevertheless, the results do indicate that the WebStore performance reaches an acceptable level, and hence that its enhanced support for program safety is not fatally marred by unacceptable performance.

6 XML and the Semantic Web

In Section 3.3 we mention that by formulating the network communication of our server in terms of XML rather than HTML, it is possible to achieve a higher level of safety as a result of the richer and more precise type system of XML (XML-Schema). In particular, the matching between XML XForms and the parameters of Web service functions can be far more precise than with HTML forms.

Clearly, a version of the WebStore that serves XML (including of course XHTML) could be constructed and would bring benefits over and above those in the current prototype (such as the aforementioned improvement in the precision of form typing). The current prototype and an XML version following the same model can both be described as *Web-enabled persistent systems*. That is, they are traditional persistent systems augmented by the exposure of a Web interface. In particular, the type system of these systems is that defined by a persistent programming language (ProcessBase in our case). Objects within this type system correspond to artifacts in the Web, and the correspondence is maintained by a bi-directional mapping process operating at the system level. As a result, artifacts such as pages are transiently untyped when represented (and possibly manipulated) in textual form. Nevertheless, as we have shown, even this limited extension to persistent systems has considerable utility: by permitting the application of persistent programming to the engineering of Web applications, these applications gain improved safety properties.

However, a more lofty goal may be in view. Given any type system of sufficient richness and strength, it is possible to engineer a persistent system for that type system. In particular, if the emerging type system of the Web (*i.e.* XML, XML Schema [11], RDF [25] *etc.*) had sufficient strength and richness to form the basis of a persistent system, then it would be possible to engineer a persistent system that safely stores values and executes computations wholly within that type system. Rather than being merely Web-enabled (as those above), this system is termed a *Web-centric persistent system*, since it takes the central notion of its design (namely its type system) from the Web. In such a system, Web artifacts remain under a single type system, independent of whether they are in textual or object graph format. This would enable, for example, the safe editing of a Web artifact in an external editor, provided that that editor complied with data type constraints for the artifact, as expressed in XML Schema and RDF.

It seems likely that the current versions of XML, XML Schema and RDF are as yet insufficiently rich and strong to satisfy our requirements (for example: XML Schema keys guarantee referential integrity within single documents, but there is no way to express integrity constraints beyond single documents; WSDL represents services by their interfaces, rather than as higher-order entities like functions in ProcessBase and the functional programming paradigm). If this is so, then the problem becomes one of extending this XML type system so that it is sufficiently rich and strong to form the basis of a Web-centric persistent system. Higher levels of the Semantic Web [26] (*e.g.* OWL [27], OWL-S [28]) are proceeding in this direction, whilst Web Services Addressing [29] is defining the ability to address Web service end points, a key pre-requisite for Web services to become first class objects.

One key problem is to support the expression of computations within this Web type system. The XML Query language [12] provides support for the expression of a limited class of computations within the XML-Schema type system. This excludes facilities for update and hence XML Query is insufficient for the implementation of a Web server. The XL project [30], has conducted a preliminary investigation into the extension of XML Query with persistent storage update facilities modelled on those of SQL. XL is, as yet, a design rather than an implementation, and so is not currently a viable means for the implementation of an XML WebStore. Furthermore, XL artificially distinguishes between updates to (persistent) data within XML documents (via update queries) and updates to transient data within running programs (via assignment). The removal of this distinction is at the heart of persistent programming, and is, we believe, one of its key simplifying benefits. It is this simplicity which enables us to implement the efficient prototype WebStore with a minimum of engineering effort.

The Xen/C-Omega work [31, 32] at Microsoft Research Labs is philosophically closer to our approach, in that the Xen/C-Omega language supports access and update of XML structures (and also relational tables) via constructs that have general application within Xen/C-Omega computations, rather than simply bolting a query/update language onto an ordinary imperative language, as in the XL design. To an extent, the Xen/C-Omega approach and ours are converging on similar goals, but from different starting points. Xen/C-Omega starts from a library-based approach (within the conventional object-oriented language C#), where any domain specific safety properties must by definition be enforced dynamically. The Xen/C-Omega language adds linguistic support for common library idioms, including static enforcement of safety properties. For example, Xen/C-Omega has a static notion of subtyping on stream types having a common element type but different arity. One result of this is that Xen/C-Omega is probably more dynamic (and hence less safe) than it strictly needs to be. Our approach, on the other hand, attempts to model Web application behaviour within a conventional type system (which is almost entirely static) via a small but carefully crafted layer of "magic" within the language run-time and HTTP interface. One consequence of this is that our approach is probably insufficiently dynamic to encompass the full scope of Web application development, with the construction of applications that cross organisational boundaries being a particular challenge. The level of dynamicity has probably reached limits determined by the use of a conventional type system, and future extension will likely involve Semantic-Web/XML specific types, such as the stream types in Xen/C-Omega.

7 Related Work

There are two broad schools of thought in relation to the problem of ensuring Web application safety. The first school advocates testing: before a Web application is put into use, it is subject to an extensive checking process which attempts to verify the various safety properties we have identified. Early Web testing tools employ exhaustive testing, which is feasible for static content but clearly infeasible for the scale of dynamic content required in large Web applications. More recent attempts use static analysis to guide the testing process, in an attempt to make it feasible to test dynamic content [33]. One major advantage of the testing approach is that it can be applied to verify properties that are not related to program safety; for example, it is possible to check that all requests complete within acceptable performance constraints.

The second school of thought on Web application safety asserts that many important properties, and specifically those identified in this paper, can be ensured statically. Our work falls within this school, and the rest of this section concentrates on approaches that provide such static guarantees.

7.1 Use of a relational database for Web content storage

Large Web sites which are concerned with preventing broken link failures in static content commonly employ a relational database for content storage. Links within the Web site are mapped to relational data governed by foreign key constraints. When a page is uploaded to the site, it is parsed into database rows, and then an attempt is made to insert these rows into the database. This attempt fails if it would violate the databases referential model as specified by foreign key constraints. Such cases arise only when one or more of the links in the page is broken, in which case the page is rejected and not published. Once a page is accepted into the database, it is served by unparsing the database rows representing the page into textual form, in much the same way as in the WebStore. This unparsing process is guaranteed not to produce broken links by virtue of the database integrity constraints, which prevent the deletion of a page representation whilst it is referenced from another page.

The main problem with this technique is that it works only for static content. When applied to dynamic content, insertion into the database occurs after the content has been requested, as the content generator executes. If a broken link is generated, which is entirely possible, then the database will reject the generated page, leading to a failure to serve the page. *I.e.* the potential failure made possible when a page containing a broken link is served is replaced by a certain failure: refusal to serve the page. In addition to this problem, we also note that this technique does not address any of the other safety properties.

7.2 W3Objects

The W3Objects system [3, 4], successfully prevents broken links from arising in static and dynamically generated content. It operates in much the same way as the WebStore, by restricting computations to operate on structured page representations, with links represented by references. In this, it may be regarded as one of the WebStore's principal precursors. W3Objects says nothing about the other three safety properties identified in this paper. In particular, it does not represent service parameters within links, and so cannot guarantee consistency between forms and services. We also note that W3Objects inherits a reference counting garbage collector from the Arjuna system [34], on which it is based. Such garbage collectors cannot collect cyclic garbage, a long term problem in the Web context since it is known that a high proportion of Web objects are involved in cycles [35].

7.3 The <bigwig> system

The `<bigwig>` system [5, 6], is probably the most comprehensive approach to Web application safety prior to our work. This system addresses three of the four safety properties we identify (well-formed HTML, consistency of links with Web services, static binding to session state) but does not consider referential integrity.

The omission of referential integrity is crucial, since it prevents integration of the safety model through induction on link properties. This leads to each safety property being considered in isolation, with significant extra complexity as the result. In particular, `<bigwig>` requires complex data-flow analyses to be performed at compilation time, and the results are strictly approximate, although this appears not to be a problem in practice.

The `<bigwig>` system exhibits a tendency to introduce new and unfamiliar constructs. They introduce a new notion of higher-order page templates, whereas in this paper, we use a pre-existing and well-understood construct, typed higher-order functions, for the same purpose. Similarly, they represent sessions as distinct programming language entities, somewhat reminiscent of Ada tasks but in other ways new and unfamiliar. Once again, we represent sessions using the pre-existing higher-order function construct.

7.4 Functional programming approaches

There is a substantial body of work on Web application development within the functional programming community [36, 37, 38, 39]. Since this work is (as one would expect) based around higher-order functions, there is significant similarity with our work. In particular, our approach to session handling can be viewed as a kind of continuation passing, and is supported as such within these systems. Updates within these functional systems are supported by generation of entirely new closures containing the updated values, then passing URLs associated with those closures back to the Web browser. WebStore differs in this case, in that it only needs to generate a new function object for the continuation, with this function's closure including the preexisting session state which is updated in place.

These systems lack an integrated model for update of shared and persistent state. For example, a multi-user Web application might maintain some variables that are shared between the sessions of several users, and subject to update from those sessions. This is difficult in functional systems, since each session is represented by an independent continuation, with no sharing of mutable state between continuations. Persistence is a further difficulty: these systems adopt a Monads based approach to interaction with the external world, including update to persistent state. This effectively places persistent data outside the safety regimes that exists within memory. Hence, the safety guarantees last only as long as the Web application is running, rather than for the full lifetime of the Web site.

7.5 IDE/Tool-based approaches - Microsoft .NET

The ASP.NET component of Microsoft's .NET framework [40] provides for the construction of Web applications supported by a range of static safety guarantees. One advantage of .NET over our system is support for multiple programming languages (*e.g.* C#, VB.Net and Eiffel) executing within the same managed environment[c].

Applications developed in .NET benefit from many of the safety properties also provided in WebStore. For example, the framework ensures safe binding between form inputs and the code that processes them. However, the means by which these safety properties are achieved is different. In .NET, the IDE tool (VisualStudio .NET) plays a key role in initialising safety properties. Applications with safety properties thus initialised are transferred to the .NET server execution environment for deployment. As one might perhaps expect, the approach reflects and leverages the proprietary nature of the Microsoft platform. In contrast, WebStore enforces safety properties wholly on the (deployment) server, with these properties being initialised as content and modules are published to the server, and maintained continuously thereafter. In the WebStore approach, developers are free to choose and use their preferred tools (*e.g.* Emacs, Dreamweaver *etc.*) to create content and program modules, with WebStore as one component within an open systems architecture.

Leaving the general controversy between open and proprietary approaches to one side, .NET's location of much of the program safety infrastructure outside the final execution environment carries significant risks:

- It may be possible for content or programs to be introduced into the execution enviroment by some route other than through the IDE. In such cases, the level of static safety guarantees are weakened.
- The ability for a executing Web application to evolve safely is reduced, since executing applications do not have access to the full range of facilities in the IDE. This is mitigated somewhat by the extensive faciltities for reflection provided in .NET, and by the fact that safe evolution is inherently hard (WebStore applications are likely to remain safe, but resist evolution).

[c].Net distinguishes between managed and un-managed applications, and the latter can be programmed in almost any language. However, enforcement of many of the static safety properties we are interested in is far more difficult in the un-managed environment, so we confine our discussion to the managed case.

The integrated, server based approach taken in WebStore averts the first risk completely and in respect of the second risk, ensures all evolution is safe. However, the gaurantee of safe evolution comes at the cost of retarding the progress of that evolution - WebStore applications are gauranteed to remain safe, but tend to resist evolution.

One final point of contrast between .NET and our approach is between library-based and language-based approaches. The .NET approach is library based, with VisualStudio having special knowledge of the Web application environment and the .NET libraries supporting it. In contrast, WebStore is language based, with Web specific knowledge contained in the system-level components of the WebStore application. It is interesting to note that Microsoft's Xen/C-Omega project (discussed previously at the end of Section 6) adopts a language based approach, which might at some point feed back into the .NET framework.

8 Conclusion

The first contribution of this paper is the presentation of a new model for the four important safety properties specifically of concern to Web application developers. A novel inductive step provides integration of the model such that the constraints enforcing the various properties become mutually supporting. This integration leads to a model that is both more general and simpler than previous work.

The second contribution is a prototype server implementing the model, the WebStore Web application server. In addition to its role as a proof-of-concept for the model, the WebStore is a compelling exemplar of the technology of orthogonal persistence on top of which it is implemented.

Finally, this paper reports preliminary performance results for the WebStore prototype. Even at this early stage these are very promising, with the WebStore marginally outperforming mainstream uni-processor Web servers for static content and demonstrating a more significant performance advantage in relation to serving dynamic content, at which the system is targeted. Whilst it would be unreasonable to extrapolate from these results into comparison with industrial strength application servers and server farms, the results do demonstrate that our approach provides performance that is acceptable for a wide class of Web applications.

Future work on the WebStore centres around the areas of *application specific scheduling*, *cacheable server pages* and *open hyper-programming*.

Application specific scheduling may be useful to enhance the ability of the WebStore to meet the soft real-time deadlines required of interactive applications. Application specific scheduling policies improve compliance to application needs since they can be formulated in terms of abstractions appropriate to the application (*e.g.* requests, involving both computation and network I/O), rather than abstractions chosen by the system software (*e.g.* threads and sockets separately). Furthermore, these policies are cognisant of specific application properties, which they may therefore exploit to the application's advantage. We are pursuing investigation of this area within the *Compliant Systems Architecture* [41, 42, 43, 24].

Server pages systems, such as JSP [44] and PHP [13], support a style of Web application construction that allows executable statements to be embedded within HTML text. Current servers have difficulty in caching server pages content, even when the majority of a given page is literal HTML. In contrast, the structured object representation used in the WebStore supports a simple approach to caching. Essentially, page fragments can be cached (in unparsed form) at the root of each static sub-tree within the representation of a given page. As a result of this inherent simplicity, implementation of caching is able to focus on efficiency.

The *hyper-programming* concept [45, 46] unifies the source and executable representation of programs. Programs under construction with this approach use pointers to bind to actual objects involved in program execution, rather than using textual access paths that are subsequently processed to bind to such objects at link time. This approach supports a wider range of static checking of program safety properties than is possible with conventional development tools. Most hyper-programming systems have been closed worlds. We see the WebStore as a vehicle to explore the hyper-programming concept within an open system, whilst also increasing the level of static program safety for Web applications. The hyper-programming concept also has the potential the provide a unified representation of Web engineering artifacts though the entire engineering life cycle, through extension to include requirements, design and test facets of a given artifact, augmenting the unified representation of source and executable code. Initial work [47] in this area has focused on supporting hyper-programming within the single server WebStore, which is open to the world (via HTTP), but which is simplified by the lack of server distribution and (more importantly) by the fact that the whole system is under a single organisational control, and hence the integrity of system level operations can be fully trusted. A truly open hyper-programming system needs to support hyper-programs that cross organisational boundaries, denying any recourse to trusted system level operations. Overcoming this constraint is a major challenge for the future.

References

1. Y. Deshpande, S. Murugesan and S. Hansen (2001), *Web engineering: Beyond CS, IS and SE Evolutionary and non-engineering perspectives*, in Web Engineering: Managing Diversity and Complexity of Web Application Development, LNCS 2016, Springer-Verlag, pp. 14–23
2. S. Murugesan, Y. Deshpande, S. Hansen and A. Ginige (2001), *Web engineering: A new discipline for development of Web-based systems*, in Web Engineering: Managing Diversity and Complexity of Web Application Development, LNCS 2016, Springer-Verlag, pp. 3–13.
3. D. Ingham, S. Caughey and M. Little (1996), *Fixing the "broken-link" problem, The W3Objects approach*, WWW5, Paris, France, `http://www5conf.inria.fr/fich_html/papers/P32/Overview.html`.
4. D. Ingham, S. Caughey, M. Little and S.K. Shrivastava (1995) *W3Objects: Bringing object-oriented technology to the Web*, WWW4, Boston, USA, `http://www.w3.org/Conferences/WWW4/Papers2/141/`.
5. C. Brabrand, A. Moller and M. Schwartzbach (2002), *The <bigwig> project*, ACM Transactions on Internet Technology, Vol. 2(2), pp. 79–114.
6. A. Sandholm and M. Schwartzbach (2000), *A type system for dynamic Web documents*, in Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language (POPL'00), pp. 290–301.
7. M.P. Atkinson, and R. Morrison (1985), *Procedures as persistent data objects*, ACM Transactions on Programming Languages and Systems, Vol. 7(4), pp. 539–559.
8. M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison (1983), *An approach to persistent programming*, Computer Journal, Vol. 26(4), pp. 360–365.
9. M.P. Atkinson, and R. Morrison (1995), *Orthogonally persistent object systems*, VLDB Journal, Vol. 4(3), pp. 319–401.
10. T. Olds, H. Detmold, K. Falkner and D.S. Munro (2004), *Engineering safe and efficient shareability within Web systems*, in Proceedings of the Sixth Asia Pacific Web Conference (APWEB'04), Hangzhou, China, LNCS 3007, Springer-Verlag, pp. 697–710.

11. World Wide Web Consortium (W3C), *XML Schema*, `http://www.w3.org/XML/Schema`.
12. World Wide Web Consortium (W3C), *XML Query*, `http://www.w3.org/XML/Query`.
13. The PHP Group, *PHP: Hypertext Preprocessor*, `http://www.php.net/`.
14. R. Jones and R. Lins. R (1996), *Garbage collection: Algorithms for automatic dynamic memory management*, John Wiley and Sons.
15. P. Wilson (1993), *Uniprocessor garbage collection techniques*, in Proceedings of the International Workshop on Memory Management, St. Malo, France, LNCS 637, Springer-Verlag, pp. 1–42.
16. World Wide Web Consortium (W3C), *XForms - The next generation of Web forms*, `http://www.w3.org/MarkUp/Forms/`.
17. R. Morrison, D. Balasubramaniam, M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro and B.C. Warboys (1999), *ProcessBase reference manual (version 1.0.6)*, Universities of Manchester and St. Andrews.
18. PS-algol (1988), *PS-algol reference manual, 4th edition*, Universities of Glasgow and St Andrews, Technical Report PPRR-12-88.
19. R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby and D.S. Munro (1996), *Napier88 reference manual (release 2.2.1)*, University of St Andrews.
20. G.N.C. Kirby (1992), *Persistent programming with strongly typed linguistic reflection*, in Proceeding of the 25th Hawaii International Conference on System Sciences (HICSS-25), Kauai, Hawaii, USA, pp. 820–831.
21. G. Trent and M. Sake, *WebStone: The first generation in HTTP server benchmarking*, `http://www.mindcraft.com/webstone/paper.html`.
22. R.B. Miller (1968), *Response time in man-computer conversational transactions*, in Proceedings of the 1968 AFIPS Fall Joint Computer Conference, San Francisco, CA, USA, Vol. 33, pp. 267–277.
23. C. Allison, M. Bramley and J. Serrano (1999), *Meeting interactive response targets in distributed learning environments*, The Active Web, Stafford: BCS.ISBN 1-897898-45-2, pp. 93–97.
24. A. Zakaravicius (2004), *Compliant Thread Scheduling*, Honours Dissertation, School of Computer Science, The University of Adelaide.
25. World Wide Web Consortium (W3C), *Resource Description Framework (RDF)*, `http://www.w3.org/RDF/`.
26. J. Hendler, T. Berners-Lee and E. Miller (2002), *Integrating applications on the semantic Web*, Journal of the Institute of Electrical Engineers of Japan, Vol. 122(10), pp. 676-680.
27. World Wide Web Consortium (W3C), *Web Ontology Language (OWL)*, `http://www.w3.org/2004/OWL/`.
28. World Wide Web Consortium (W3C), *OWL-S: Semantic markup for Web services*, `http://www.w3.org/Submission/OWL-S/`.
29. World Wide Web Consortium (W3C), *Web Services Addressing working group*, `http://www.w3.org/2002/ws/addr/`.
30. D. Florescu, A. Grunhagen and D. Kossmann (2002), *XL: An XML programming language for Web service specification and composition*, WWW2002, Honolulu, Hawaii, USA, `http://www2002.org/CDROM/refereed/481/`.
31. E. Meijer, W. Schulte and G. Bierman (2003), *Programming with circles, triangles and rectangles*, in Proceedings of XML2003.
32. E. Meijer and W. Schulte (2003), *Unifying tables, objects and documents*, in Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Programming (DP-COOL'03), Uppsala, Sweden, pp. 145–166.
33. M. Benedikt, J. Freire and P. Godefroid (2002), *VeriWeb: Automatically testing dynamic Web sites*, WWW2002 Alternate Paper Track, Honolulu, Hawaii, `http://www2002.org/CDROM/alternate/654/index.html`.
34. S.K. Shrivastava, G.N. Dixon and G.D. Parrington (1991), *An overview of the Arjuna distributed programming system*, IEEE Software, Vol. 8(1), pp. 66–73.
35. N. Richer and M. Shapiro (2000), *The memory behaviour of the WWW, or the WWW considered as a persistent store*, in Proceedings of the 9th International Workshop on Persistent Object

Systems (POS-9), Lillehammer, Norway, LNCS 2135, Springer-Verlag, pp. 161–176.

36. P. Graunke, S. Krishnamurthi, S. Van Der Hoeven and M. Fellesien (2001), *Programming the Web with high-level programming languages*, in Proceedings of the 10th European Symposium on Programming (ESOP'01), LNCS 2028, Springer-Verlag, pp. 122–136.
37. M. Hanus (2001), *High-level server-side scripting in Curry*, in Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01), LNCS 1990, Springer-Verlag, pp. 76–92.
38. C. Queinnec (2000), *The influence of browsers on evaluators, or continuations to program Web servers*, in Procedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM SIGPLAN Notices, Vol. 35(9), pp. 23–33.
39. P. Thiemann (2005), *An embedded domain-specific language for type-safe server-side Web scripting*, ACM Transactions on Internet Technology, Vol. 5(1), pp. 1–46.
40. Microsoft Corporation, *MSDN library: Overview of the .NET framework (.NET framework developer's guide)*, `http://msdn.microsoft.com/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp`.
41. R.M. Greenwood, K. Mayes, B.C. Warboys, B.S. Yeomans, D. Balasubramaniam, G.N.C. Kirby and R. Morrison (2000), *System evolution, feedback and compliant architectures*, in Proceedings of the International Workshop on Feedback and Evolution in Software and Business Processes (FEAST 2000), Imperial College, London, UK.
42. R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro and B.C. Warboys (2000), *An approach to compliance in software architectures*, Computing and Control Engineering, Special Issue on Informatics, Vol. 11(4), pp. 195–200.
43. R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro and B.C. Warboys (2000), *A compliant persistent architecture*, Software, Practice & Experience, Vol. 30(4), pp. 363–386.
44. Sun Microsystems, *Java Server Pages technology, version 1.1*, `http://java.sun.com/products/jsp/`.
45. R. Morrison, R.C.H. Connor, Q.I. Cutts, A. Dearle, A. Farkas, G.N.C. Kirby, R. McGettrick and E. Zirintsis (1999), *Current directions in hyper-programming*, in Proceedings of the 3rd International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI), Novosibirsk, Russia, LNCS 1755, Springer-Verlag, pp. 316–340.
46. E. Zirintsis, G.N.C. Kirby and R. Morrison (1999), *Demonstration of Hyper-Programming in Java*, in Proceedings of the 25th International Conference on Very Large Databases (VLDB'99), Edinburgh, Scotland, pp. 734–737.
47. J.D. Fox, H. Detmold and K.E. Falkner (2004). *Hyper-programming Web applications*, in Proceedings of the Second Annual Australian Undergraduate Students' Computing Conference, Melbourne, Australia, pp. 59–66.