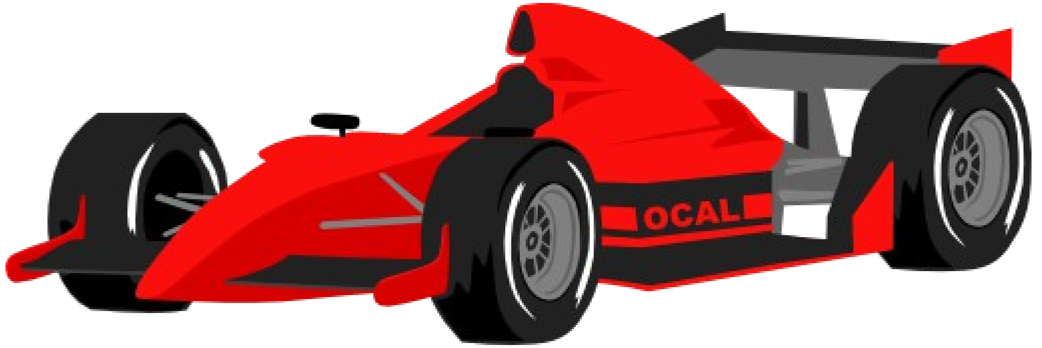
An Optimizing Compiler for a Purely Functional Web-Application Language

Adam Chlipala – MIT CSAIL
ICFP 2015
August 31, 2015

Choosing a PL for a Web App



the eternal question

Performance

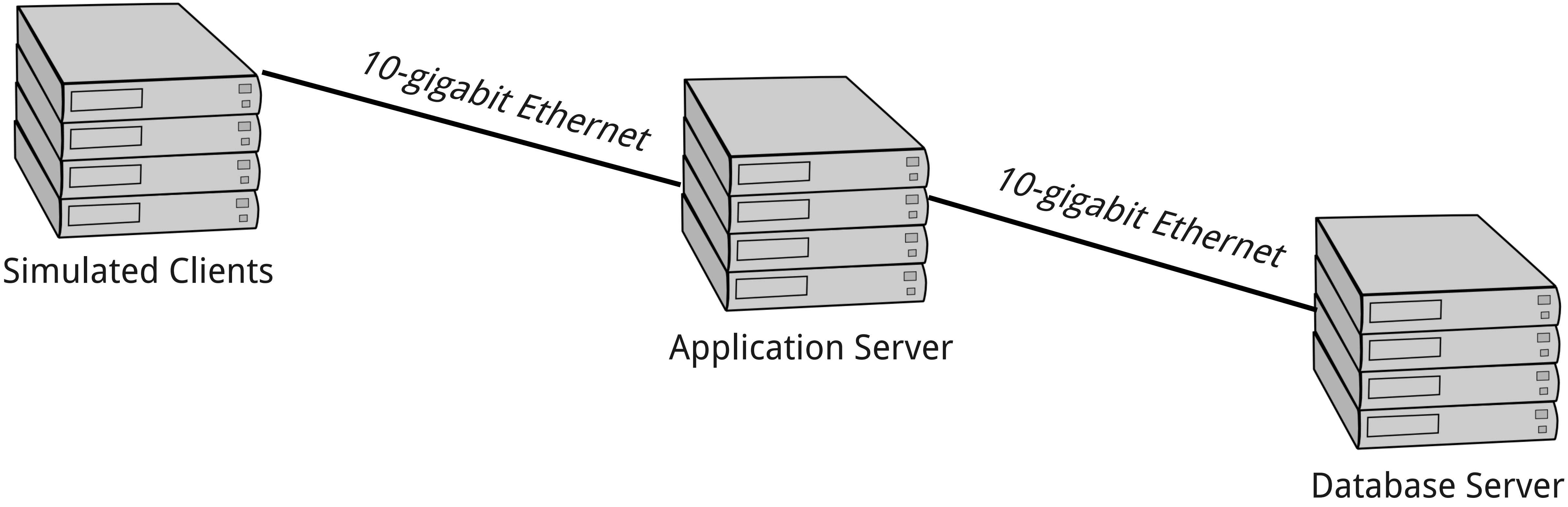
Productivity

Exemplars:
C, C++
(e.g., the “are you serious?” bloc)

Exemplars:
JavaScript, Ruby, Python
(e.g., the “better buy a few more servers” bloc)

Benchmarking Web Apps

<http://www.techempower.com/benchmarks/>



Each machine has the same hardware:
32 GB of RAM plus <u>40 hyperthreads</u>
(so implementations with weak concurrency stories will fall behind)

Which framework should we choose?

“all JavaScript, all the time”

Hype Meter

Ur/Web: A Functional DSL for the Web

```
(** A new table, specific to this test *)
table fortune : {Id : int, Message : string} PRIMARY KEY Id

(** Here's the additional fortune mandated by the spec. *)
val new_fortune =
    {Id = 0, Message = "Additional fortune added at request time."}

(** Actual page handler *)
fun fortunes () =
  fs <- queryL1 (SELECT fortune.Id, fortune.Message FROM fortune);
  fs' <- return (List.sort (fn x y => x.Message > y.Message) (new_fortune :: fs));
  return <xml>
    <head><title>Fortunes</title></head>
    <body><table>
      <tr><th>id</th><th>message</th></tr>
      {List.mapX (fn f => <xml><tr>
        <td>{[f.Id]}</td><td>{[f.Message]}</td>
      </tr></xml>) fs'}
    </table></body>
  </xml>
```

- Purely functional
- Rich (almost dependent) type system
- Monads
- Type classes
- ML-style modules

Peeking at the Node.js Manual

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

Take a URL string, and return an object.

Pass `true` as the second argument to also parse the query string using the `querystring` module. If `true` then the `query` property will always be assigned an object, and the `search` property will always be a (possibly empty) string. Defaults to `false`.

Pass `true` as the third argument to treat `//foo/bar` as `{ host: 'foo', pathname: '/bar' }` rather than `{ pathname: '//foo/bar' }`. Defaults to `false`.

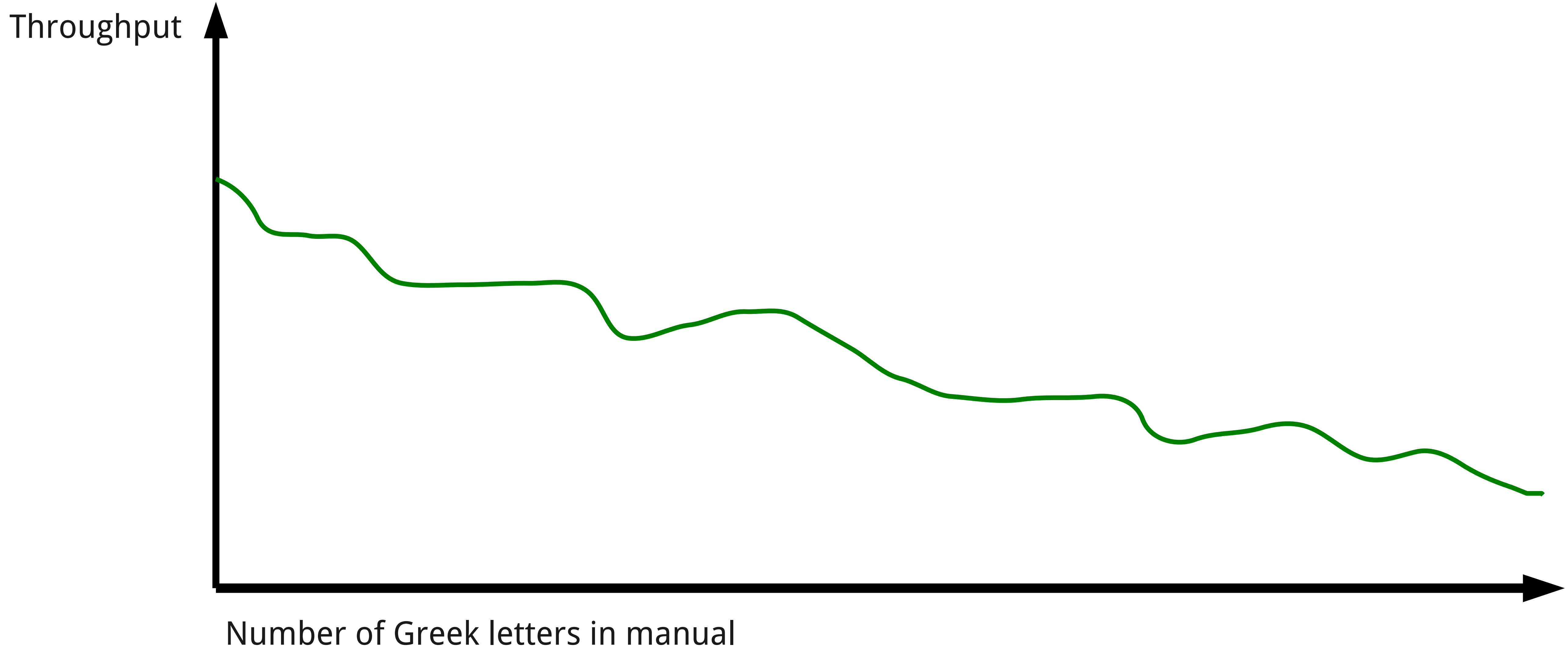
Peeking at the Ur/Web Manual

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau : \tau} \quad \frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \ell : T(\ell)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \mathcal{I}(\tau)} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ x) = \tau}{\Gamma \vdash M.x : \mathcal{I}(\tau)} \quad \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \mathcal{I}(\tau)} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ X) = \tau}{\Gamma \vdash M.X : \mathcal{I}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \Rightarrow e : \tau_1 \to \tau_2}$$

Let's Do Some Science



The Envelope, Please?

<http://www.techempower.com/benchmarks/>[*]

Best (bar chart) | Data table | Latency | Framework overhead

Best fortunes responses per second, Dell R720xd dual-Xeon E5 v2 + 10 GbE

Framework	Best performance (higher is better)		Cls	Lng
ulib-postgres	321,889	100.0%	Plt	C++
urweb-postgres	299,675	93.1%	Ful	Ur
ulib-mysql	183,583	57.0%	Plt	C++
gemini-postgres	162,484	50.5%	Ful	Jav
compojure	158,950	49.4%	Mcr	Clj
cpoll_cppsp-raw	151,581	47.1%	Plt	C++
⋮				
nodejs	56,487	17.5%	Plt	JS
⋮				
yesod	25,421	7.9%	Ful	Hkl

[*]Data from Round 11 Preview 2 – official release should be out soon!

Message for the Rest of the Talk:

“You can do this at home.”

The Ur/Web compiler follows a conceptually straightforward optimization strategy that you, too, can apply, with relatively little effort, to compile your functional program so that it routinely trounces C++ code in performance.

*Caveat: it's essential to use a *domain-specific language* where the compiler can be informed about the deep semantics of the operations that programs perform!

The Most Important Decision

Use a whole-program compiler.
(inspired by the MLton Standard ML compiler)

After type checking,
flatten all module structure,
eliminate all abstraction barriers,
and **inline** all uses of functors (module functions).

Example Program Again

```
(** A new table, specific to this test *)
table fortune : {Id : int, Message : string} PRIMARY KEY Id

(** Here's the additional fortune mandated by the spec. *)
val new_fortune =
    {Id = 0, Message = "Additional fortune added at request time."}

(** Actual page handler *)
fun fortunes () =
  fs <- queryL1 (SELECT fortune.Id, fortune.Message FROM fortune);
  fs' <- return (List.sort (fn x y => x.Message > y.Message) (new_fortune :: fs));
  return <xml>
    <head><title>Fortunes</title></head>
    <body><table>
      <tr><th>id</th><th>message</th></tr>
      {List.mapX (fn f => <xml><tr>
        <td>{[f.Id]}</td><td>{[f.Message]}</td>
      </tr></xml>) fs'}
    </table></body>
  </xml>
```

Specializing Definitions, in several flavors

```
datatype list a =
    Nil
  | Cons of a * list a

fun sort [a] (f : a -> a -> bool) (ls : list a)
  : list a = …


sort (fn x y => x.Message > y.Message) ls
```

Step 1. **Unpoly**

Specializing Definitions, in several flavors

```
datatype list a =
    Nil
  | Cons of a * list a

fun sort' (f : T -> T -> bool) (ls : list T)
  : list string = …


sort' (fn x y => x.Message > y.Message) ls
```

Step 2. **Specialize**

Specializing Definitions, in several flavors

```
datatype list' =
    Nil'
  | Cons' of T * list'

fun sort' (f : T -> T -> bool) (ls : list')
  : list' = …


sort' (fn x y => x.Message > y.Message) ls
```

Step 3. **Especialize**

(call-pattern specialization)

Specializing Definitions, in several flavors

```
datatype list' =
    Nil'
  | Cons' of T * list'

fun sort'' (ls : list')
  : list' = …


sort'' ls
```

Unveiling Abstractions & Going Impure

```
<tr><th>id</th><th>mess
{List.mapX (fn f =>
  <td>{[f.Id]}</td><t
```

```
th = fn x => “<th>” ^ x ^ “</th>”
```

really means (in simplified/styli

```
concat (tr (concat (th (cdata “id”),
                    th (cdata “message”)),
        List.mapX (fn f => tr (concat (td (cdata (show f.Id)),
                                       td (cdata f.Message)))) ls)
```

Embedded-language syntax desugars into combinator calls.

Step 4. **Monoize**

(translate to monomorphic, impure language & expose definitions of combinators)

Unveiling Abstractions & Going Impure

```
“<tr>” ^ “<th>” ^ escape “id” ^ “</th>”
       ^ “<th>” ^ escape “message” ^ “</th>” ^ “</tr>”
^ List.mapX (fn f => “<tr>” ^ “<td>” ^ escape (show f.Id)
                     ^ “</td>” ^ “<td>” ^ escape f.Message
                     ^ “</td>” ^ “</tr>”) ls
```

Step 5. **Reduce**

(algebraic simplification)

Unveiling Abstractions & Going Impure

```
“<tr><th>id</th><th>message</th></tr>”
^ List.mapX (fn f => “<tr><td>” ^ escape_int f.Id
                     ^ “</td><td>” ^ escape f.Message
                     ^ “</td></tr>”) ls
```

Taking Advantage of Side Effects

```
write("<tr><th>id</th><th>message</th></tr>"
  ^ List.mapX (fn f => "<tr><td>" ^ escape_int f.Id
                        ^ "</td><td>" ^ escape f.Message
                        ^ "</td></tr>") ls);
```

Actually, **Monoize** compiles code to insert explicit `write()` operations, sending strings to the browser imperatively.

Step 5. **Reduce** (again)

(algebraic simplification)

Taking Advantage of Side Effects

```
write(“<tr><th>id</th><th>message</th></tr>”);
write(List.mapX (fn f => “<tr><td>” ^ escape_int f.Id
                       ^ “</td><td>” ^ escape f.Message
                       ^ “</td></tr>”) ls));

fun mapX f ls =
   case ls of
       Nil => “”
     | Cons (x, ls') => f x ^ mapX f ls'
```

Taking Advantage of Side Effects

```
write(“<tr><th>id</th><th>message</th></tr>”);
write(mapX' ls);
```

```
fun mapX' ls =
   case ls of
       Nil => “”
     | Cons (x, ls') => “<tr><td>” ^ escape_int x.Id
                        ^ “</td><td>” ^ escape x.Message
                        ^ “</td></tr>” ^ mapX' ls'
```

Step 6. **Fuse**

(push `write()` inside recursive function definitions)

Taking Advantage of Side Effects

```
write("<tr><th>id</th><th>message</th></tr>");
mapX'' ls;
```

```
fun mapX'' ls =
   case ls of
       Nil => write("")
     | Cons (x, ls') => write("<tr><td>" ^ escape_int x.Id
                           ^ "</td><td>" ^ escape x.Message
                           ^ "</td></tr>" ^ mapX' ls')
```

Step 6.5. **Reduce** (again)

(algebraic simplification)

Taking Advantage of Side Effects

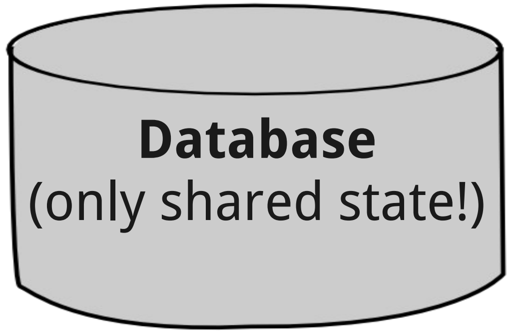
```
write(“<tr><th>id</th><th>message</th></tr>”);
mapX'' ls;
```

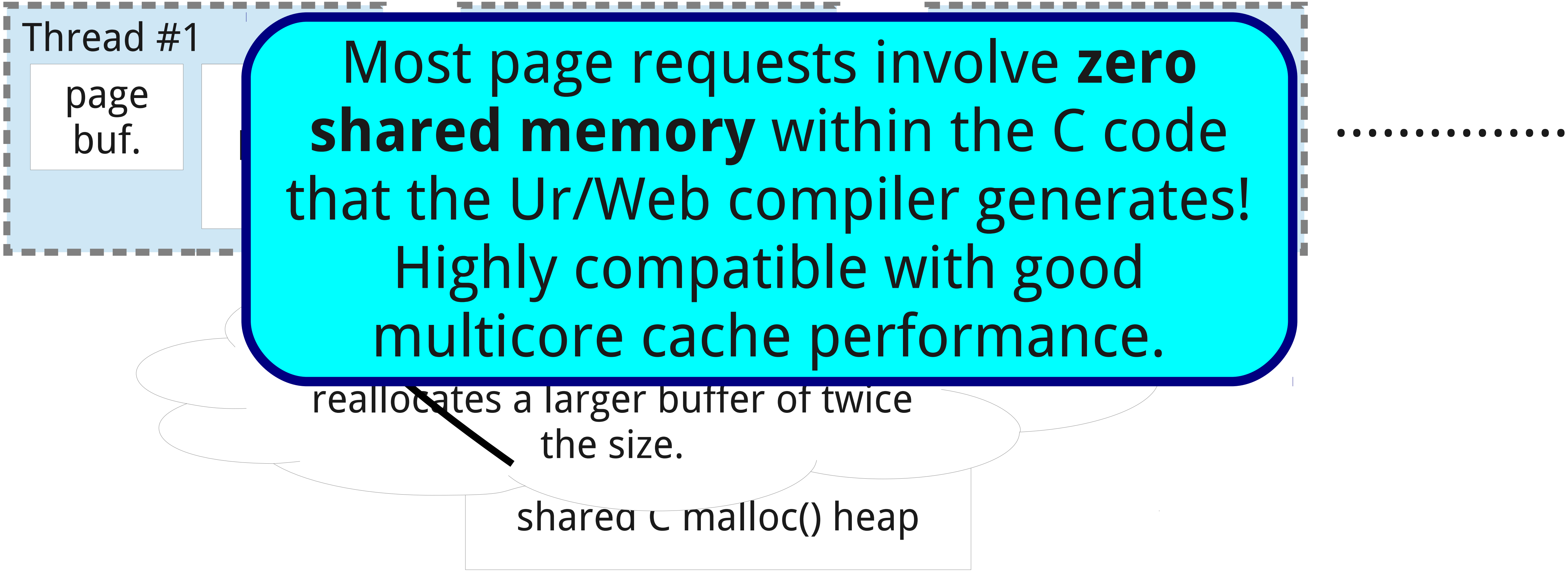
```
fun mapX'' ls =
   case ls of
       Nil => ()
     | Cons (x, ls') => write(“<tr><td>”); escape_int_w x.Id;
                        write(“</td><td>”); escape_w x.Message;
                        write(“</td></tr>”); mapX'' ls'
```

Mission Accomplished! 

Zero allocation: we write directly into an imperative page buffer.

Memory Management





Most page requests involve **zero shared memory** within the C code that the Ur/Web compiler generates! Highly compatible with good multicore cache performance.

That Looks Too Easy....

How do you do garbage collection?

- **Transactions** are integrated into Ur/Web at a deep level, so, whenever we run out of space, we can always *abort* the execution, allocate a larger heap, and restart.
- As a further optimization, we use **region-based memory management**, inferring a stack structure to allow freeing whole sets of objects at key points during execution.

In Summary

A simple compilation strategy makes it possible to compile programs from
    a purely functional language based on dependent type theory
to
    some of the fastest web-application servers on the planet
    (e.g., 300k requests/sec. in benchmark, beating ~100 popular frameworks)

No dataflow analysis
No control-flow analysis
No garbage collector
(though we compile via GCC, which provides some of the above later in the pipeline)

Use this strategy for your next functional DSL!

Open source at:
http://www.impredicative.com/ur/

One Last Domain-Specification Optimization

```
x = runQuery(“SELECT foo.Title FROM foo WHERE foo.Id = ” ^ id);
```

Step 7. **Prepare**

(find opportunities to infer SQL prepared statements, allowing *advance query compilation*)

One Last Domain-Specification Optimization

```
q1 = prepare(“SELECT foo.Title FROM foo WHERE foo.Id = ?”);


x = runPrepared(q1, [id]);
```