MetaOCaml Server Pages:

Web publishing as staged computation

Christopher League

NEPLS
27 October 2005

Web site = computer program

- Modern dynamic web services are computer programs
  - To support collaboration & personalization
  - Examples: web mail, e-commerce, 'blogs, event calendar, political action network, etc.

Performance matters

- A dramatic increase in web traffic can bring down the server (the *slashdot* effect)

Gospel according to CmdrTaco

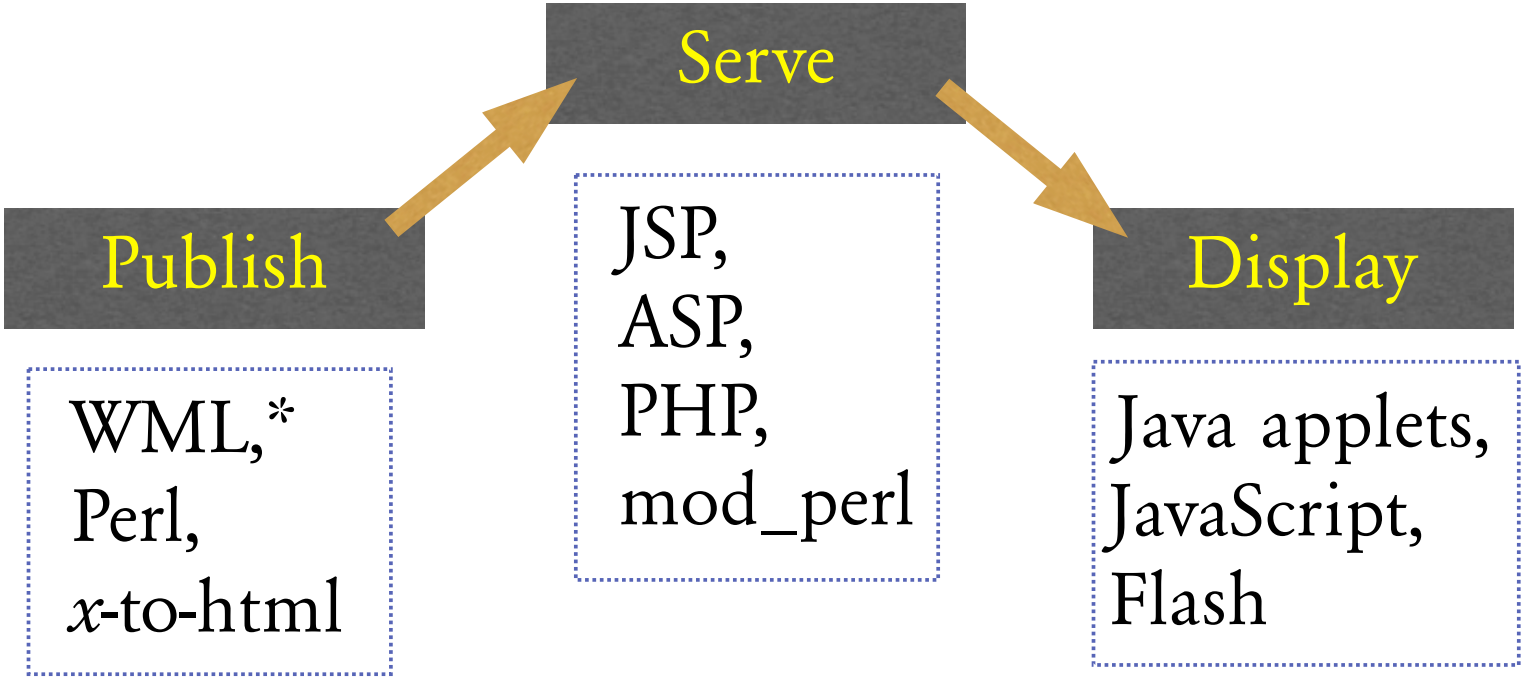
> "When you're actually loading a page, even if it's a complicated page that looks dynamic and custom, we're really just putting together a bunch of puzzle pieces that have been pre-generated, and making the simplest, quickest decisions we possibly can."
>
> — Rob Malda, *creator of slashdot*

from J. Turner, "How to survive being slashdotted." *LinuxWorld Magazine* 2(1), 2003.

3 stages of web service

1. Developer publishes content
2. Server transfers content
3. Browser displays content

Each stage, different language



*Web-site Meta Language –
"off-line HTML generation toolkit"

Staging using today's tools

- One script outputs another
  - Values passed from one stage to the next as strings
  - Programmer manages quoting and cross-stage persistence by hand

Staging using today's tools

- Example: unholy PHP code for cross-stage persistence:

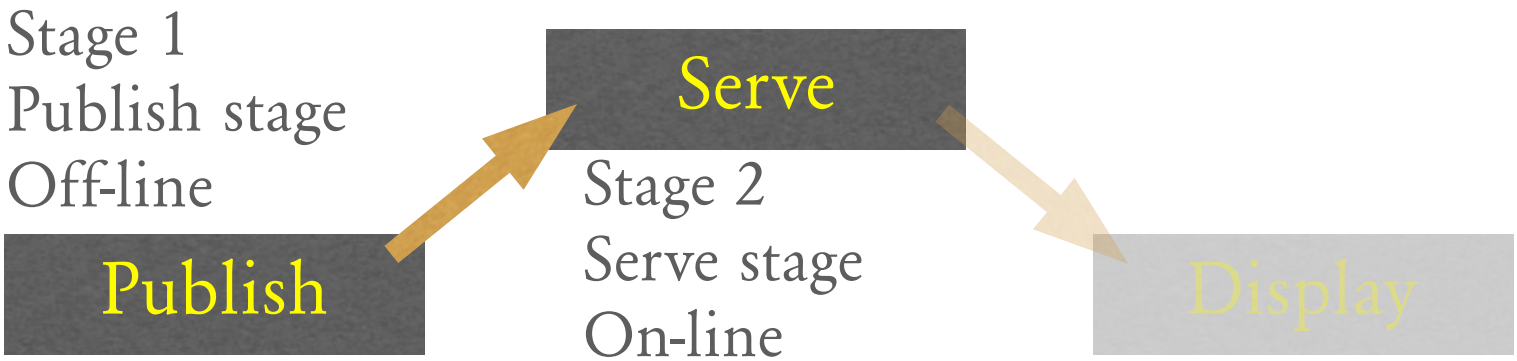
```
<?= "<?\n" ?>
<?= "\$data = unserialize(\"".
     addcslashes(serialize($data),'"').
     "\");\n" ?>
<?= "?>\n" ?>
```

Our idea

- A single web programming language that can express various staging possibilities, safely and precisely,
  - ➙ by leveraging the staging annotations of MetaOCaml.

    [Calcagno, Taha, Huang, & Leroy: GPCE '03]

Caveat



- We exclude the final (display) stage from our system, for now.

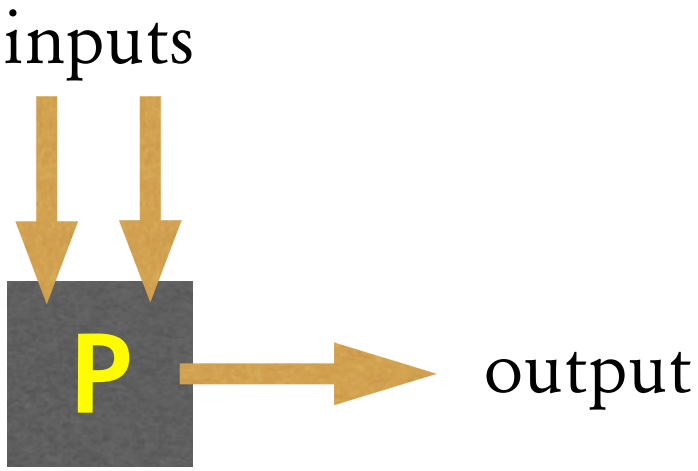
Outline

- Review of multi-stage language
- Design of MetaOCaml Server Pages
- Examples & demonstration
- Performance results
- Limitations & future work

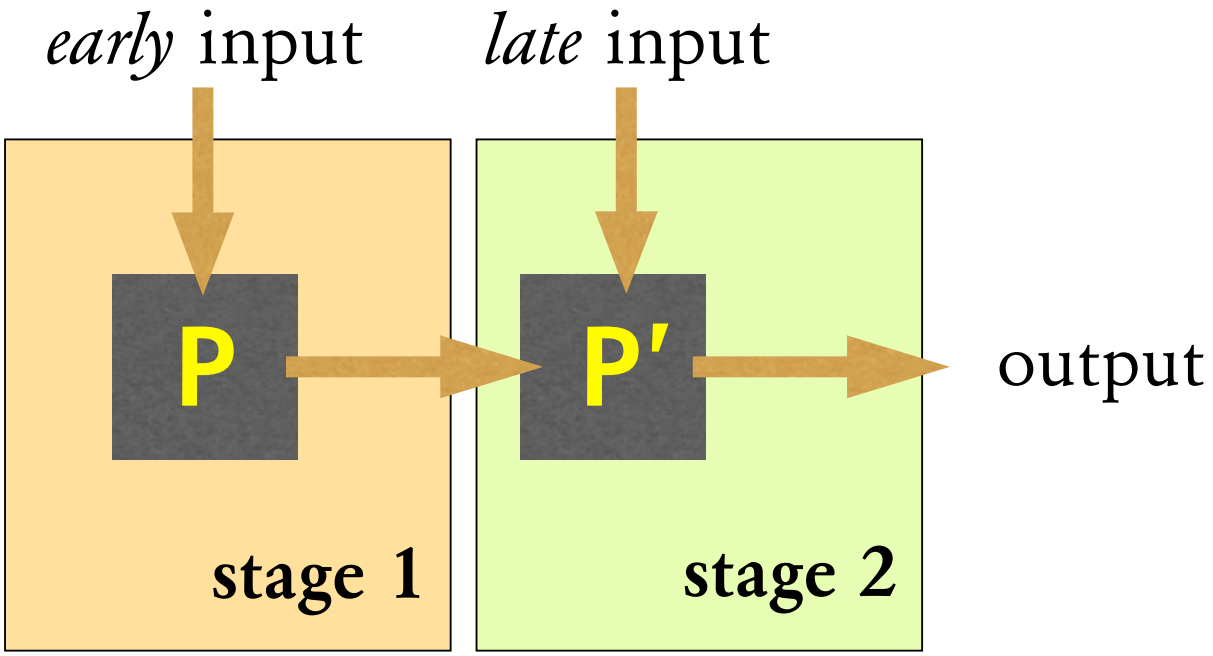
What is multi-stage prog?

- Type-safe program generation
  - One program produces another program as its output
  - The output program can be executed some time later, possibly many times.

Unstaged computation



Staged computation



Staging annotations: MetaOCaml

.< *expr* **>.**	brackets
.~ *expr*	escape
.! *expr*	run

‘Brackets’ construct code

- Normally, expressions are evaluated immediately:

$$3 * 4 \rightarrow 12$$

- Brackets cause the expression within to be delayed until some future stage:

$$.< 3 * 4 >. \not\rightarrow$$

‘Run’ executes code

.! .< 3 * 4 >. → 3 * 4 → 12

‘Escape’ splices in code

```
.< 3 * .~(.< 4 * 5 >.) >.  →
.< 3 * (4 * 5) >.
```

- Programs annotated with these operators are capable of generating custom code to be executed later.

‘Escape’ is not delayed

.< 3 * .~(**let** y = 4 * 5 **in** .< y >.) >. →

.< 3 * .~(**let** y = 20 **in** .< y >.) >. →

.< 3 * .~(.< 20 >.) >. →

.< 3 * 20 >.

Typical example: power function

```
let even n = (n mod 2) = 0
let square x = x * x
(* power : int → int code → int code *)
let rec power n x =
      if n = 0 then .< 1 >.
      else if even n then
          .<square .~(power (n/2) x)>.
      else
          .<.~x * .~(power (n-1) x)>.
```

Typical example: power function

```
.! .<fun x → .~(power 11 .<x>.)>.
⇒ fun x → x * square(x * square(square x))

let rec power n x =
    if n = 0 then .< 1 >.
    else if even n then
        .<square .~(power (n/2) x)>.
    else
        .<.~x * .~(power (n-1) x)>.
```

Outline

- ✔ Review of multi-stage language
- Design of MetaOCaml Server Pages
- Examples & demonstration
- Performance results
- Limitations & future work

‘Server page’ conventions

- Source is text/html by default.
- Embed code between delimiters:

```
<h1>This is text</h1>
<? puts “And this is code.” ?>
```

Various kinds of code blocks

- Declarations – evaluated in publish stage, but also lifted above other code

```
<?^  open Queue
     let some_function x y = ... ?>
```

- Serve-stage code

```
<? let result = some_function a b ?>
```

- Short-cuts for printing strings

```
<?= string_of_int result ?>
<?"%4d" result ?>
```

Translating a server page

- Before they may be used, the server page syntax must be translated to plain MetaOCaml.

Translating a server page

```
<? pragma args a b c ?>
<? ^ declarations ?>
<? statements ?>
<?= string_to_be_printed ?>
Regular text.
<? "format string" d, e ?>
<? ^ more_declarations ?>
<? let x = expression ?>
<? more_statements ?>
Bye!
```

```
module Trans = struct
let lift x = .<x>.
 declarations
 more_declarations
let page a b c = .<fun req puts →
let arg = Request.arg req in
 statements ;
puts ( string_to_be_printed );
puts "Regular text.\n";
Printf.kprintf puts "format string" ( d) ( e);
 let x = expression in
 more_statements ;
puts "Bye!\n";
>.
end
```

Syntactic sugar for staging

- Use '~' to splice in publish-stage code.

```
<?~ a ?>          →  <? .~(a) ?>
<?~= b ?>         →  <?= .~(b) ?>
<?~let x = c ?>   →  <?let x = .~(c) ?>
```

- Use '!' to execute in publish stage.

```
<?! d ?>          →  <? .~(lift(d)) ?>
<?!= e ?>         →  <?= .~(lift(e)) ?>
<?!let x = f ?>   →  <?let x = .~(lift(f)) ?>
```

Outline

- ✔ Review of multi-stage language
- ✔ Design of MetaOCaml Server Pages
- Examples & demonstration
- Performance results
- Limitations & future work

(switch to demo)

Staged power script

```
<?^ open Num (* for arbitrary-precision arithmetic *)
  let width = 54
  let rec wrap puts s = (* wrap 's' into a fixed-width block *)
    if String.length s ≤ width then puts s else
      (puts (Str.string_before s width); puts "\n";
       wrap puts (Str.string_after s width))
  let is_zero = eq_num (Int 0)
  let square x = .< let z = .~x in z */ z >.
  let rec power n x = (* staged power function *)
    if is_zero n then .< Int 1 >. else
    if is_zero (mod_num n (Int 2)) then square(power (n//Int 2) x)
    else .< .~x */ .~(power (n -/ Int 1) x) >. ?>
<? pragma args y ?>
<?! let y' = string_of_num y ?>
<? let x' = match (arg "x") with Some v → v | None → "2" ?>
<?= preamble(x'^"^"^y') (* Output begins here *) ?>
<?!= navbar("/power"^string_of_num y) ?>
<form method='get'> This page computes
  <input name='x' type='text' value='<?= x' ?>' size='20'/>
  <sup><?= y' ?></sup> </form>
<?~ let result = power y .< num_of_string x' >. ?>
<p>The result is:
<pre><? wrap puts (string_of_num result) ?></pre></p>
<?= postamble ?>
```

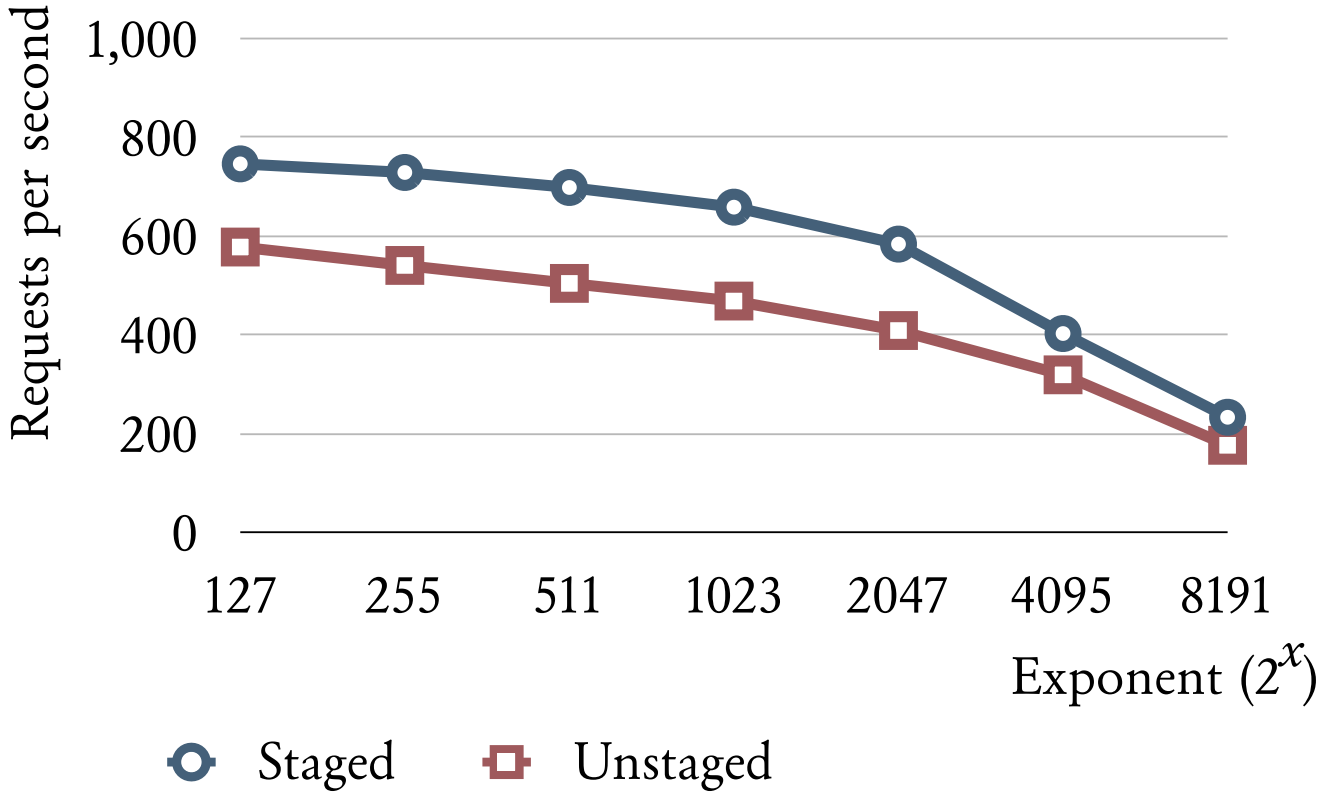
Outline

- ✔ Review of multi-stage language
- ✔ Design of MetaOCaml Server Pages
- ✔ Examples & demonstration
- Performance results
- Limitations & future work

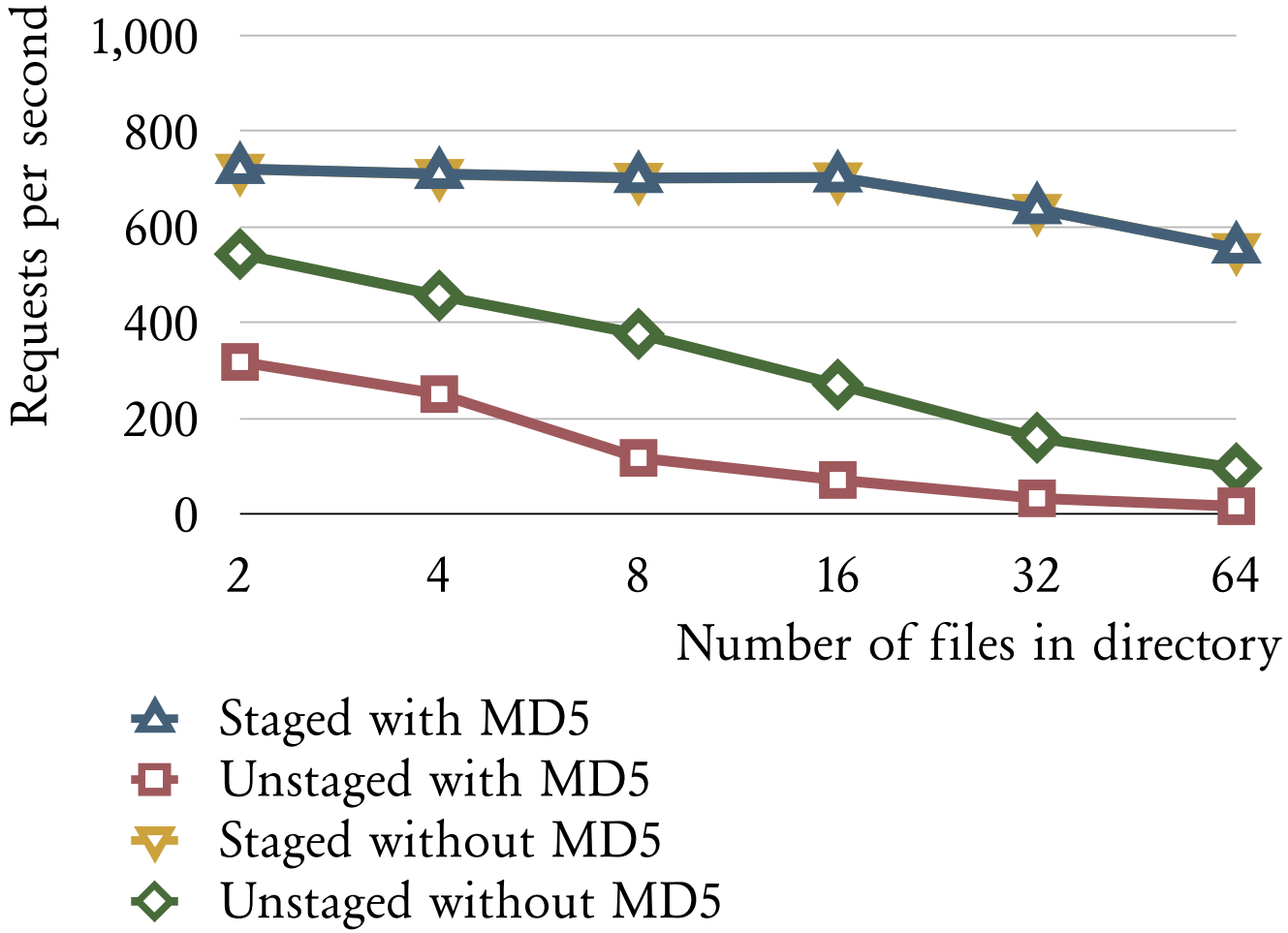
Methodology

- Measured throughput—number of requests answered per second
- Apache HTTP benchmarking tool (ab) issued requests from 8 threads simultaneously for 30 seconds
- On otherwise idle Intel Xeon workstation: Linux 2.6, 768MB RAM, 512kB cache, Ultra160 SCSI

Throughput for power function



Throughput for dir. browsing



Outline

- ✔ Review of multi-stage language
- ✔ Design of MetaOCaml Server Pages
- ✔ Examples & demonstration
- ✔ Performance results
- • Limitations & future work

Limitations

- MetaOCaml cannot (yet!) read/write generated code from/to disk.
- Therefore, all server pages must be available in memory when server starts.
- Error messages refer to translated code, not the source.

Future directions

- Extend to display (third) stage.
- Statically validate generated (X)HTML.

  [Wallace & Runciman: ICFP '99]
  [Elsman & Larsen: PADL '04]
- Stage a complete content management system (CMS)
- Implement as module of a real server (*e.g.*, Apache).

Conclusion

- MetaOCaml server pages:
  a new domain-specific language for web applications programming.
- Provides safe and precise control over staging of web services.
- Substantial benefits in performance and expressiveness.