

Dynamic Software Updating

Michael Hicks

Department of Computer Science
Cornell University

Non-stop Systems

Must run without interruption

- ◆ Telephone switches
- ◆ Financial transaction processors
- ◆ Air traffic control systems
- ◆ Internet servers (e.g. For e-commerce)

... but require upgrades

For the purpose of

- ◆ bug fixes
- ◆ functionality/performance enhancements

Example: VISAnet

- ◆ VISA credit card approval system
- ◆ Tolerates less than 0.5% downtime per year
- ◆ but requires up to 20,000 code updates per year

Flawed Approach

Stop and redeploy with new software

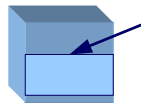
- ◆ But cancels current processing
- ◆ Loses accumulated state

Not acceptable for mission critical apps

Flawed Approach

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
```

Start: existing source



current state

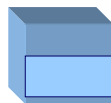
Running system

Flawed Approach

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

Start: existing source

Modify program as needed



Running system

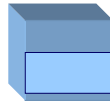
Flawed Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source
Modify program as needed
Compile it and test it



New version

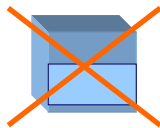


Running system

Flawed Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source
Modify program as needed
Compile it and test it
Halt existing system
cancels current processing
loses system state



Flawed Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source
Modify program as needed
Compile it and test it
Halt existing system
Start new version

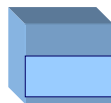


Running system

Our Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c

Start: existing source



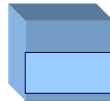
Running system

Our Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source

Modify program as needed



Running system

Our Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source

Modify program as needed

Compile it and test it



New version

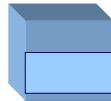


Running system

Our Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source
Modify program as needed
Compile it and test it
Develop dynamic patches



Running system

Our Approach

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Start: existing source
Modify program as needed
Compile it and test it
Develop dynamic patches
Apply patches to running system



Running system

Our Approach

Key elements:

- ◆ Builds on type-safe dynamic linking of native code
- ◆ Library implementation of updating API
- ◆ Generates dynamic patches mostly automatically
- ◆ Informed by actual use in updateable server app

FlashEd Webserver

- Case study: the Flash webserver, but Editable
 - ◆ Roughly 8,000 lines of code
- Built incrementally
- Version 0.1 deployed publicly Oct 12, 2000
 - ◆ <http://flashed.cis.upenn.edu/>
- Three major updates since then

Timeline



Oct 12

FlashEd v1

Timeline



Oct 12

Oct 27

FlashEd v1

FlashEd v2

- ◆ Added pathname translation caching (FlashEd v2)

Timeline



FlashEd v1

FlashEd v2

- ◆ Added pathname translation caching (FlashEd v2)

FlashEd v3

- ◆ Added 32 MB file cache

Timeline



FlashEd v1

FlashEd v2

- ◆ Added pathname translation caching

FlashEd v3

- ◆ Added 32 MB file cache

FlashEd v4

- ◆ Added directory listing

DSU Evaluation

Flexibility

- ◆ no system-imposed restrictions on update form or timing

Robustness

- ◆ simple implementation
- ◆ patches verifiably safe
- ◆ automation reduces errors in update code

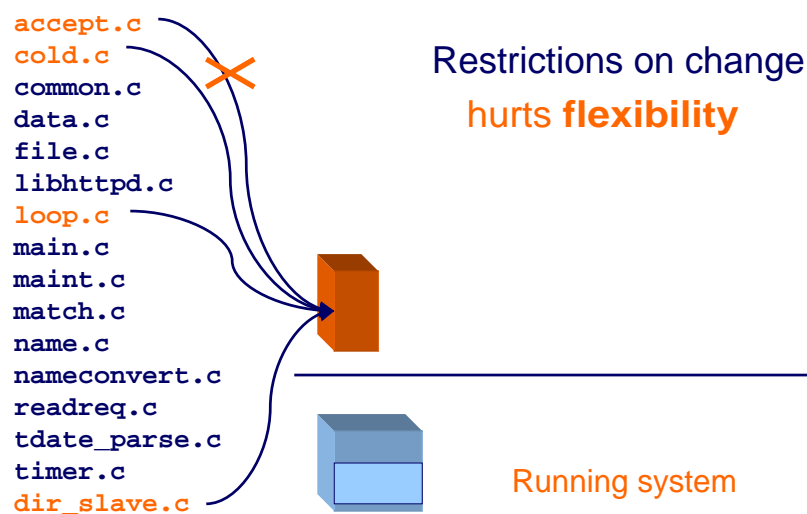
Low overhead

- ◆ less than 1% under typical conditions for sample app

Ease of use

- ◆ fits in typical software development process

Previous Approaches



Previous Approaches

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Restrictions on change
May crash the system

hurts **robustness**



Running system

Previous Approaches

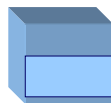
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

?



Restrictions on change
May crash the system
Hand patch generation

hurts **robustness** and
ease-of-use



Running system

Previous Approaches

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Restrictions on change
May crash the system
Hand patch generation
High overhead

hurts performance



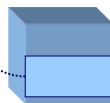
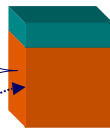
Running system

Previous Approaches

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

Restrictions on change
May crash the system
Hand patch generation
High overhead
Extra engineering

hurts robustness



Running system

DSU Implementation

Source programs in Popcorn

- ◆ Verifiably-safe (i.e. type-safe)
- ◆ Procedural (i.e. C-like)
- ◆ May define named types (i.e. C **structs**)
- ◆ No threads

Compiled to Typed Assembly Language (TAL),

- ◆ variant of Proof-Carrying Code (PCC)
- ◆ Implementation for Intel IA32 architecture

Strategy

- Dynamic patch: the unit of update
- Enable a program to be dynamically patched
- Mostly automatically generate dynamic patches
- Make sure updates are well-timed

Dynamic Patch

Reflects changes to per-module

- ◆ code
- ◆ state
- ◆ type definitions

Patch has form (f,S) where

- ◆ f is the new module
- ◆ S is a **state transformer** function to migrate old state to the new implementation

Dynamic Patches

```
f
static int num = 0;
typedef struct {
    int a; int b;
} t;
int f(t T) {
    num++;
    return T.a+T.b;
}
```

Dynamic Patches

f

```
static int num = 0;
typedef struct {
    int a; int b;
} t;
int f(t T) {
    num++;
    return T.a+T.b;
}
```

f' (new f)

```
static int num = 0;
typedef struct {
    int a; int b;
    int c;
} t;
int f(t T) {
    num++;
    return T.a*T.b*T.c;
}
```

Dynamic Patches

f

```
static int num = 0;
typedef struct {
    int a; int b;
} t;
int f(t T) {
    num++;
    return T.a+T.b;
}
```

f' (new f)

```
static int num = 0;
typedef struct {
    int a; int b;
    int c;
} t;
int f(t T) {
    num++;
    return T.a*T.b*T.c;
}
```

state transformer

Dynamic Patch

```
void init() {
    f'::num = f::num;
}
```


Strategy

- Dynamic patch: the unit of update
- Enable a program to be dynamically patched
- Generate dynamic patches mostly automatically
- Make sure updates are well-timed

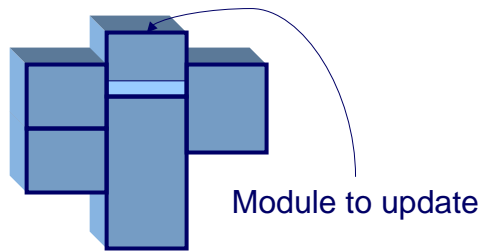
Implementing Patching

Based on dynamic loading/linking

- ◆ Dynamically load and verify the patch
- ◆ “Fix up” the running program to use the new code
- ◆ Run the state transformer
- ◆ Proceed with the program

Implementing Patching

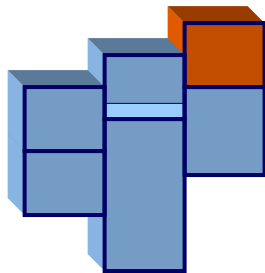
A running program



Implementing Patching

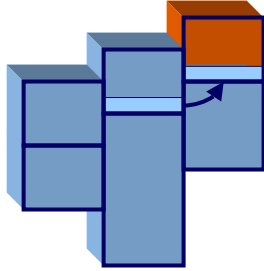
A running program

▪Load patch



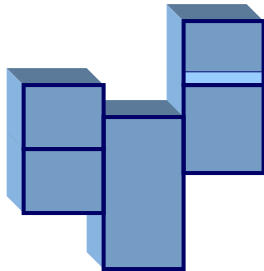
Implementing Patching

- Load patch
- Run state transformer

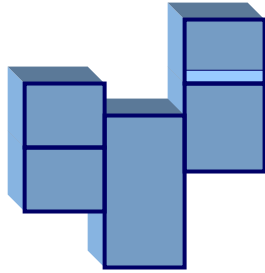


Implementing Patching

- Load patch
- Run state transformer
- Fix up program



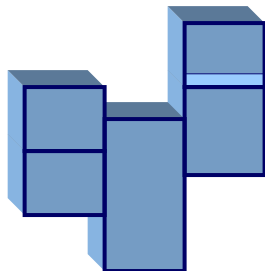
Implementing Patching



- Load patch
- Run state transformer
- Fix up program

•Can test off-line

Implementing Patching



- Load patch
- Run state transformer
- Fix up program

•Can test off-line
•Old code runs with new

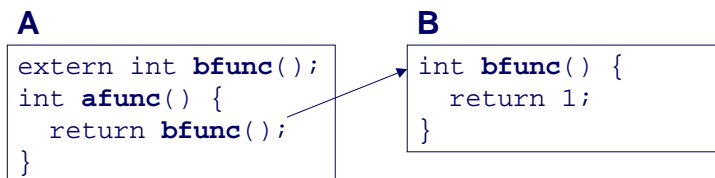
Enabling Dynamic Updates

Updating code and data. Two possibilities

- ◆ **Code relinking**
- ◆ Reference Indirection

Updating type definitions ...

Code Relinking



Before

Code Relinking

A

```
extern int bfunc();  
int afunc() {  
    return bfunc();  
}
```

B

```
int bfunc() {  
    return 1;  
}
```

new B

```
int bfunc() {  
    return 2;  
}
```

After

A

```
extern int bfunc();  
int afunc() {  
    return bfunc();  
}
```

B

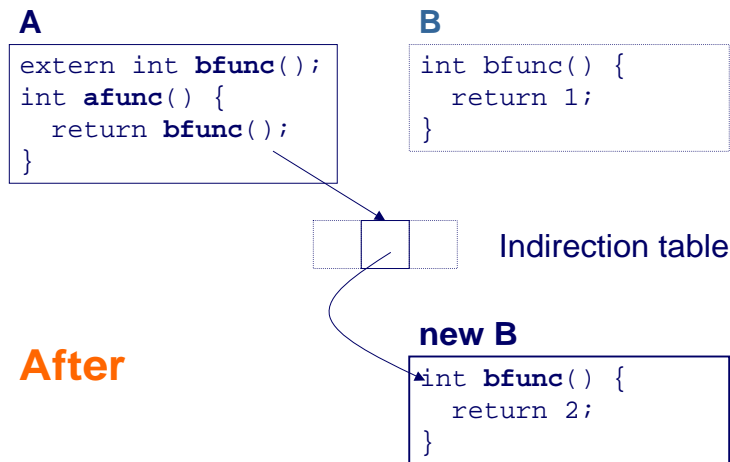
```
int bfunc() {  
    return 1;  
}
```



Indirection table

Before

Reference Indirection



Relinking vs. Indirection

Performance

- ◆ No extra indirection

Complexity (Robustness)

- ◆ Must track existing code to relink it
- ◆ But ensures all clients of global data are updated

Flexibility

- ◆ Does not deal with pointerful data; must use state transformer

Enabling Dynamic Updates

Updating type definitions. Two choices:

- ◆ Type replacement
- ◆ **Type renaming**

Updating Type Definitions

Type Replacement. Replace existing definition with new one; requires

- ◆ updating the type-checking context;
- ◆ updating all instances of the old type;
- ◆ user must update all clients of the old type.

Type Renaming.

- ◆ When a type changes, rename it.

Type Replacement

A

```
t aelem =
```

```
t a = 1
```

B

```
typedef struct t {  
  int a;  
} t;
```

Before

typechecking context

```
t → struct { int a; }
```

Type Replacement

A

```
t aelem =
```

```
t a = 1  
  b = 0
```

B

```
typedef struct t {  
  int a;  
} t;
```

After

typechecking context

```
t → struct { int a; }  
t → struct {  
  int a;  
  int b;  
}
```

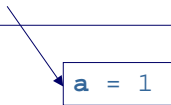
new B

```
typedef struct t {  
  int a;  
  int b;  
} t;
```

Type Renaming

A

```
t aelem =
```



```
a = 1
```

B

```
typedef struct t {  
    int a;  
} t;
```

Before

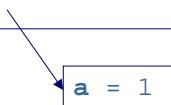
typechecking context

```
t → struct { int a; }
```

Type Renaming

A

```
t aelem =
```



```
a = 1
```

B

```
typedef struct t {  
    int a;  
} t;
```

After

typechecking context

```
t → struct { int a; }  
t2 → struct {  
    int a;  
    int b;  
}
```

new B

```
typedef struct t2 {  
    int a;  
    int b;  
} t2;
```

Renaming vs. Replacement

Flexibility

- ◆ Type renaming does not allow replacing old types

Robustness

- ◆ Replacement requires a complex implementation
- ◆ Renaming good enough; much simpler

Strategy

- Dynamic patch: the unit of update
- Enable a program to be dynamically patched
- Generate dynamic patches mostly automatically
- Make sure updates are well-timed

Generating Patches

Development Process for Updateable Software

- ◆ Make changes
- ◆ Identify which files changed and how
- ◆ Construct patches

Generating Patches

Development Process for Updateable Software

- ◆ Make changes
 - ◆ Identify which files changed and how
 - ◆ Construct patches
- Can do (mostly) automatically**
- ◆ Benefits to Robustness and Ease-of-Use

Automatic Patch Generation

- Compares files syntactically, informed by types
- Identifies and acts on changed definitions
 - ◆ For **types**, a new name is generated (MD5 hash of definition) and the mapping noted. **Type conversion function** also created
- Generates state transformer function
 - leaves placeholder when action not known

FlashEd Patches

to version	total patches	patch LOC	
		auto	by hand
0.2	16	1324	48
0.3	14	1261	99
0.4	12	1214	99

By hand patch generation: 6.5% total LOC

Strategy

- Dynamic patch: the unit of update
- Enable a program to be dynamically patched
- Generate dynamic patches mostly automatically
- Make sure updates are well-timed

Controlling Update timing

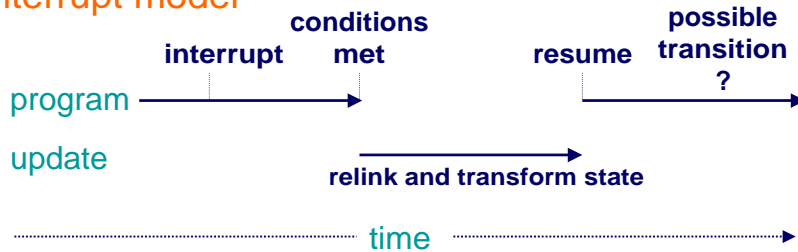
Properly time patch to avoid race conditions

Two models:

- ◆ Interrupt model
- ◆ Invoke model

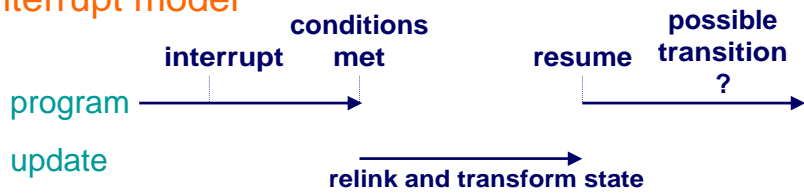
Update Timing

Interrupt model

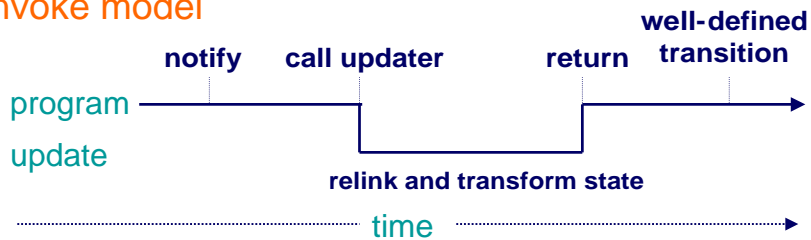


Update Timing

Interrupt model



Invoke model



Comparing Models

Interrupt model enforces timing at runtime

- ◆ Complicates the implementation
- ◆ Hard to use correctly

Invoke **enforces timing at development time**

- ◆ Applications perform their own updating. Simpler implementation, more assurances of correctness

FlashEd Update Timing

Structure amenable to invoke model

- ◆ Toplevel event loop
- ◆ Updates occur as events

Many server applications structured in this way

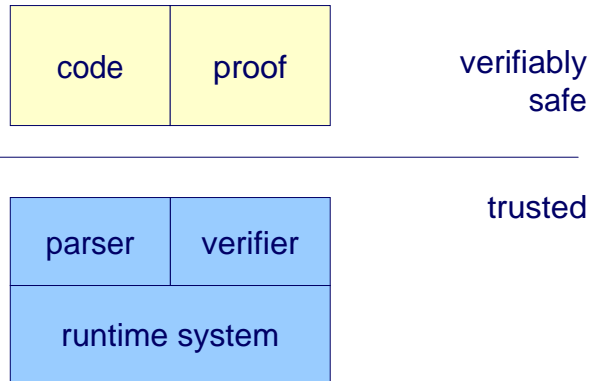
Summary of Approach

- Dynamic loading of patches
 - ◆ Patches are verified as safe before they are linked
- Code/data updates by code relinking
- Type updates by type renaming
- Mostly automatic generation of patches
- Timing ensured at software development time

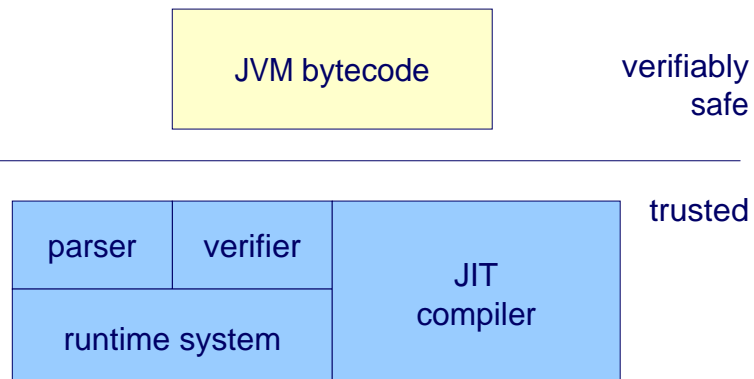
Implementation Summary

- Trustworthy **dynamic linking** for TAL
- Popcorn **source-to-source translation** to enable linking and updating
- Popcorn **library** for updating; based on C Dlopen

Verifiable Native Code

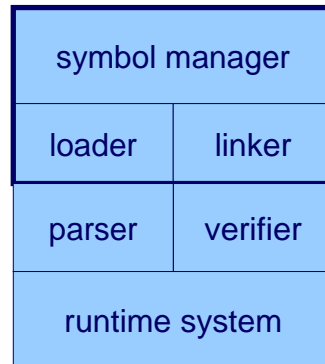


Compare to JVM



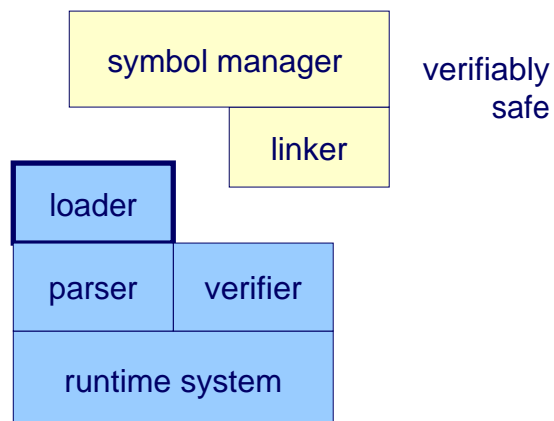
Dynamic Linking in VNC

naïve
approach

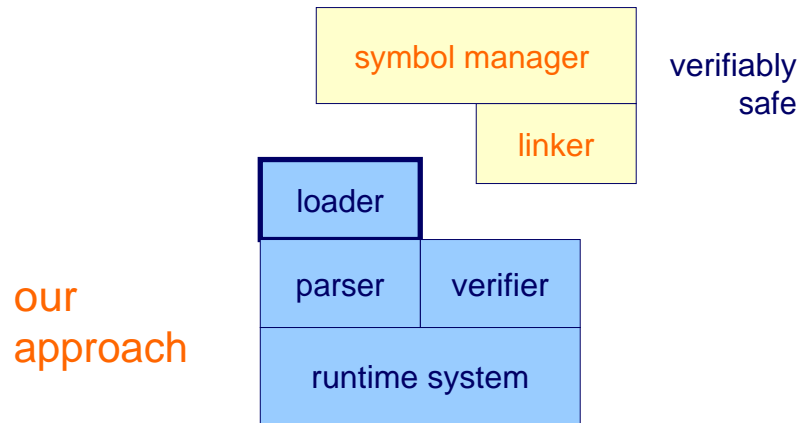


Dynamic Linking in VNC

our
approach



Dynamic Updating in VNC



Trusted Part

load primitive

- ◆ performs loading and verification

To implement linking and symbol management:

- ◆ runtime term representations for types
- ◆ **checked_cast** primitive
- ◆ existential types

Untrusted Part

Files **compiled** to have:

- ◆ **indirection table** (GOT) for external references

Library implements program interface:

- ◆ Initiates loading, unloading, linking, and updating
- ◆ Manages type-safe dynamic symbol-table

Updating Performance

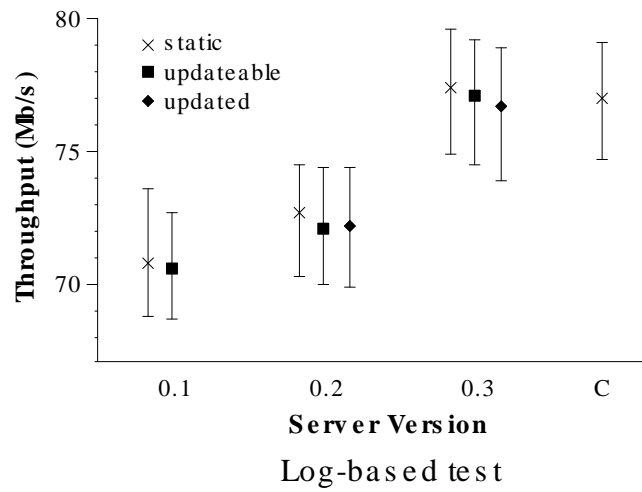
Run-time overhead

- ◆ One extra indirection per external reference, inherited from dynamic linking
- ◆ Loaded code stored in the heap; may result in more frequent GC and different cache locality

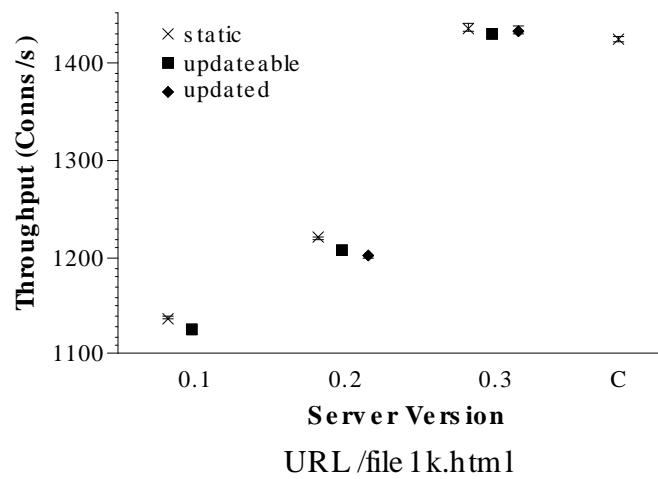
Load-time overhead

- ◆ TAL verification and dynamic linking

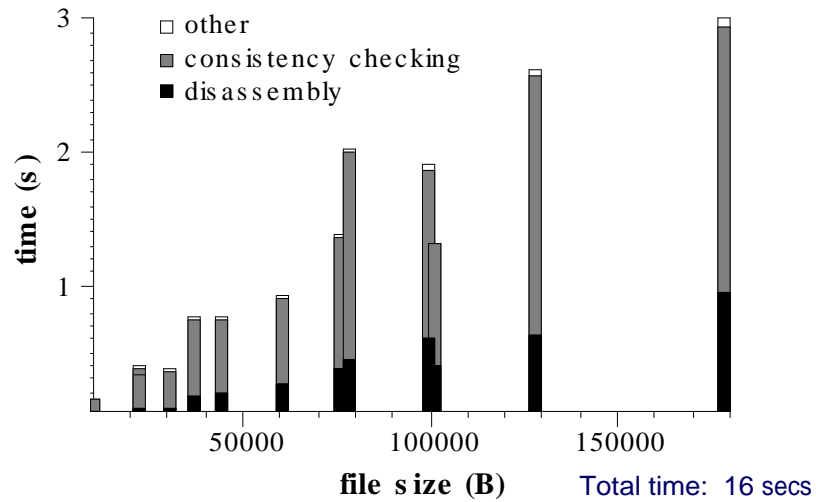
FlashEd Performance



FlashEd Performance



Loading/ Verification Times



Related Work

Updating by state transfer

- ◆ PolyLith, Gupta et al, Argus, ...
- ◆ Checkpointing and general-purpose persistence

Updating by dynamic linking

- ◆ Dynamic ML, Dynamic C++, Dynamic Java, ...
- ◆ Extensible systems and 'adaptive software'
- ◆ Active networks

Future Work

New contexts

- ◆ Functional & object-oriented languages
- ◆ Distributed systems
 - ◆ Software Routers

New features

Formalize update timing constraints

Conclusions

Developed a general-purpose dynamic updating system that is

- ◆ Flexible
- ◆ Robust
- ◆ Low overhead
- ◆ Easy to use

Validated by a realistic, non-stop application

For more information

Project Homepage

<http://www.cis.upenn.edu/~mwh/flashed.html>

- Source code available
- See thesis or papers in PLDI 2001, TIC 2000, IWAN 2000

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.