Parallel & Concurrent Programming:
Server Architectures and Beyond

Emery Berger
CMPSCI 691W
Spring 2006
Outline

- Last time:
  - Server architectures
    - Focus: web servers
    - Performance & ease of programming
  - Result – event-driven + helpers seems “better”

- Today:
  - Can we have our cake and eat it, too?
    - Where “cake” = performance + ease of programming
Server Architectures

Recap:

- **MT/MP**
  - Context switch overhead
  - Race conditions, etc.

- **SPED**
  - High throughput, but complex
  - Blocking I/O

- **AMPED**
  - Better than SPED, but still hard to program
Are Threads Just Broken?

- Events too hard:
  Can we fix threads instead?
- More natural abstraction, but:
  - Scalability limit
    - Stack size problem:
      2 MB per stack = 1000 thread limit
  - No admissions control (à la SEDA)
  - Still stuck with potential races…
“Scalable Threads for Internet Services” [von Behren et al. SOSP 2003]
- Compiler-supported approach
- User-level only
  - “For now”
- Introduces linked stacks & resource-aware scheduler
Linked Stacks

- Uses control-flow graph & compiler-inserted “checkpoints” (?) to dynamically allocate stack chunks
- Point stack pointer to new chunk before function call
- Function exit: chunk on free list
- Library code?
  - Programmer-supplied annotations...
Resource-Aware Scheduling

- Capriccio identifies stages by computing blocking graphs
  - Edge between consecutive blocking points
  - Built at runtime (nodes = call chains)
  - Tracks runtime, resource usage
- Schedules nodes to maximize utilization
  - Throttle back: schedule nodes that release resources
Scheduling: Details

- Separate run queues for each node
  - Determine priority based on predicted resource needs of node, overall utilization
  - Performs **stride scheduling**
    [Waldspurger & Weihl 95]
    - Assigns **tickets** to nodes
    - **Stride** inversely proportional to #tickets
      - = wait time until next time scheduled
- Tracks CPU, memory consumption, # file descriptors
- Result: doesn’t quite work
Scalability Results
Scalability with I/O

![Graph showing scalability with I/O](image)
Pain of Architectures

- Events hard, threads still suck
- Must commit up front
  - Difficult to change
  - New, better architecture = rewrite code
- Difficult:
  - To program & understand
    - Interleave server logic with runtime
  - Identify bottlenecks
  - Predict performance before deployment
Flux: DSL for Servers

- **Ease-of-use**
  - Declarative, implicitly parallel

- **Reuse**
  - Use unmodified code

- **Runtime Independence**
  - Not tied to any model
    - Thread-based, thread pool, event-driven

- **Correctness**
  - No deadlock!

- **Performance Prediction**

- **Bottleneck Analysis**
Flux Servers

- To date, we have built four servers in Flux:
  - Web server
  - BitTorrent “peer”
  - Image scaling server
  - Game server (“tag”)
- Very concise language
Flux Example

Page (int socket) => ();
ReadRequest (int socket)
   => (int socket, bool close, char* request);
Reply (int socket, bool close, int length, char* content, char* output) => ();
ReadWrite (int socket, bool close, char* file)
   => (int socket, bool close, int length, char* content, char* output);
Listen () => (int socket);

source Listen => Page;
Page = ReadRequest => ReadWrite => Reply;
handle error ReadWrite => FourOhFor;
handle error ReadRequest => BadRequest;
Flux Language

- **Concrete Nodes**
  - Correspond to C/C++ implementations
  - Type signatures: outputs follow inputs
    - **ReadRequest**
      - Parses client input
    - **Compress**
      - Compresses images
    - **Write**
      - Outputs compressed image to client

```plaintext
ReadRequest (int socket) -> (int socket, char* data);
Compress (int socket, char* raw, int size) -> (int socket, char* jpeg, int size);
Write (int socket, char* data, int size) -> (int socket)
```
**Flux Language**

- **Source nodes**
  - Concrete nodes that only produce output
  - Execute inside *infinite loop*

```source
source Listen => Image;
```

- **Listen**
  - Transfers control to `Image` whenever receives connection
**Flux Language**

- **Abstract Nodes**
  - Captures flow across nodes

```plaintext
Image =
    ReadRequest -> CheckCache -> Handler
    -> Write -> Complete;
```

- **Image**
  - Checks cache for requested image, handles result, writes output, and completes
Flux Language

- **Error Paths**
  - If error occurs in node, transfer to error handler
  - Ex: file not found = 404 error

```plaintext
handle error ReadInFromDisk -> FourOhFour;
```
**Flux Language**

- **Predicate Types**
  - “semantic types” in draft
  - Boolean function applied to node’s output
    - Ex: `hit` means `TestInCache` returned true when applied to argument

```c
typedef hit TestInCache;

Handler: [_, _, hit] = ;
Handler: [_, _, _] =
  ReadInFromDisk -> Compress
  -> StoreInCache;
```
Flux Language

- **Concurrency constraints**
  - *Labels* indicate which nodes cannot execute simultaneously
  - *Readers/Writers*: readers append “?”
    - Default – writers (“!”)
- **Session constraints**
  - Only applied to particular “sessions”

```plaintext
constraint CheckCache:{cache};
constraint StoreInCache:{cache};
constraint Complete:{cache};
```
Concurrency Constraints

- Safety
  - Compiler enforces canonical lock order
  - Reentrant
    - No multiple locking bugs
- Data flows acyclic
  - No deadlock!
  - Note: loops inside implementations & implicitly from client
Concurrency Constraints

- Efficiency
  - Exposing constraints lets compiler generate code specialized for different runtimes
  - Multithreaded – generate locks
  - SPED – no locks required
Complete Flux Program

- Automatically generates graphical representation
- BitTorrent example
  - Arcs denote flow
  - Annotations denote types
Web Server Performance

![Graph showing throughput vs simultaneous clients for different server models.](image)
BitTorrent Performance

![Graph showing BitTorrent performance with throughput in Mb/s against simultaneous clients. The graph compares Ctorrent, Pure Threaded (Flux), Thread Pool (Flux), and Event (Flux).]
Performance Prediction

- Generates discrete event simulator
- Uses parameters from uniprocessor run
The End