



## *Advancing High-Performance Applications on Linux Itanium*

<http://gelato.uwaterloo.ca>

### **SMP Concurrent Software Development**

**Peter Buhr (faculty)**  
**Richard Bilson (staff)**  
**Ashif Harji (Ph.D.)**  
**Roy Krischer (Ph.D.)**  
**Justyna Gidzinski (M.Math)**  
**Ayelet Israeli (M.Math)**

*Slides 1, 2, 3, 4*

### **High-Performance Web Servers**

**Tim Brecht (faculty)**  
**Mark Groves (staff)**  
**Gary Yeung (M.Math)**

*Slides 5, 6, 7, 8*

### *Research Sponsors*



# $\mu$ C++: Concurrency in C++

- C++ has no concurrency;  $\mu$ C++ provides high-level, integrated, object-oriented concurrency
- based on 3 fundamental properties: thread, stack, and mutual exclusion/synchronization (MES)
  - stack/ thread via **coroutine/ task**, extensions of **class**, and statements **suspend/ resume**
  - MES via **mutex/ nomutex** qualifier and statements **wait/ signal/ accept**

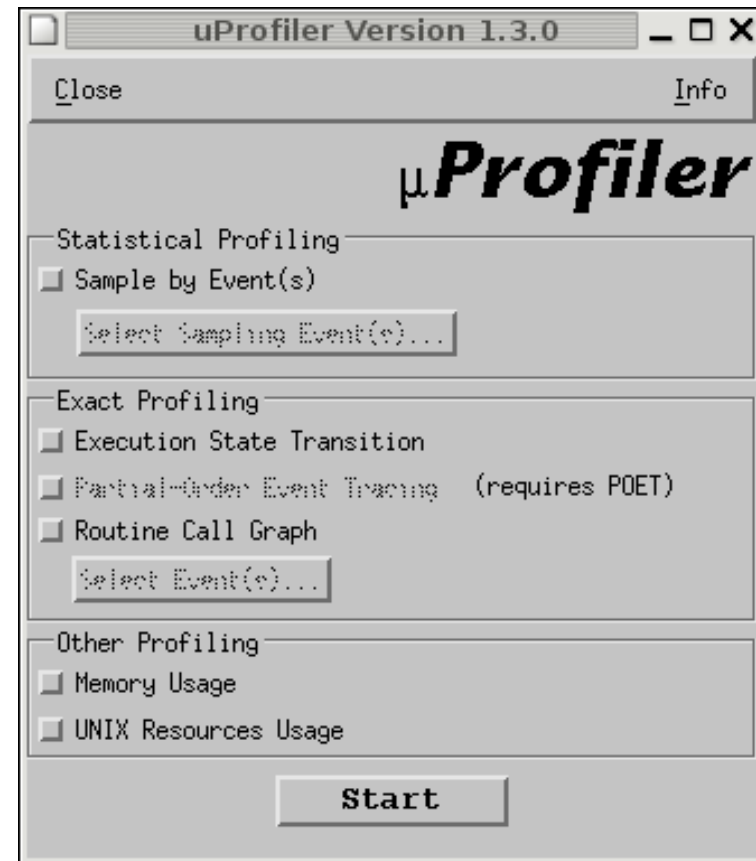
- M:N thread model (versus 1:1 NPTL)
- polymorphism: inheritance, overloading and templates with new types
- advanced concurrent exception handling : exceptions among coroutines/tasks
  - throw** [ *throw-event* [ **at** *coroutine/task* ] ] ; // *termination*
  - resume** [ *resume-event* [ **at** *coroutine/task* ] ] ; // *resumption*
- real-time: extensible schedulers, inheritance protocol, timeout, real-time tasks: **periodic, aperiodic, sporadic**
- object-oriented, nonblocking I/O for files and sockets
- user-defined grouping of tasks and *virtual* processors
- Pthreads simulation and access to OpenMP
- debug mode for testing (asserts and runtime checks)
- ports: gcc-3.4.x or greater, or **Intel icc 8.1/9** for **Linux IA-64 (HP/SGI)**, Linux Opteron, Linux/FreeBSD IA-32/AMD, Solaris 8/9 SPARC, IRIX 6.x MIPS
- <http://plg.uwaterloo.ca/~usystem/uC++.html>

	No MES	MES
No Stack / No Thread	<pre>class c { public:   m() {} };</pre>	<pre>mutex class M {   condition variables; public:   m() { wait/signal/accept } };</pre>
Stack / No Thread	<pre>coroutine C {   main() { suspend } public:   m() { resume } };</pre>	<pre>mutex coroutine CM {   condition variables;   main() { suspend/wait/signal/accept } public:   m() { resume/wait/signal/accept } };</pre>
Stack / Thread	not supported ⇒ explicit locking	<pre>task T {   condition variables;   main() { suspend/wait/signal/accept } public:   m() { resume/wait/signal/accept } };</pre>

# $\mu$ Profiler: Profiling in $\mu$ C++

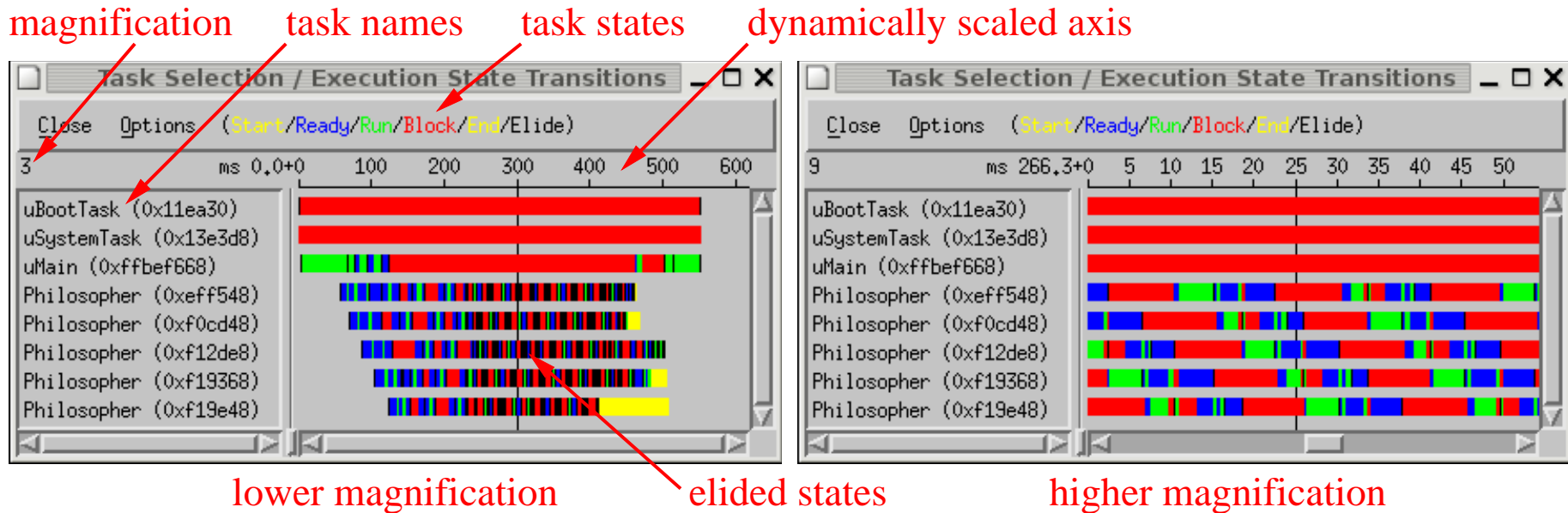
- information about dynamic execution
- integrated with  $\mu$ C++ programming model
  - break down per-task, per-coroutine, per-routine
  - trace multiple concurrent events
  - effective, efficient and extensible
- **supports hardware event-counters**
- built-in metrics
  - statistical:
    - \* dynamic call-graph, execution time, hardware counters
  - exact:
    - \* state transition
    - \* dynamic call-graph, execution time, hardware counters
    - \* partial-order tracing
    - \* memory leaks
    - \* kernel-threads

- ports: **Linux IA-64 (HP/SGI)**, Linux IA-32/AMD, Solaris 8/9 SPARC
- <http://plg.uwaterloo.ca/~usystem/MVD.html>



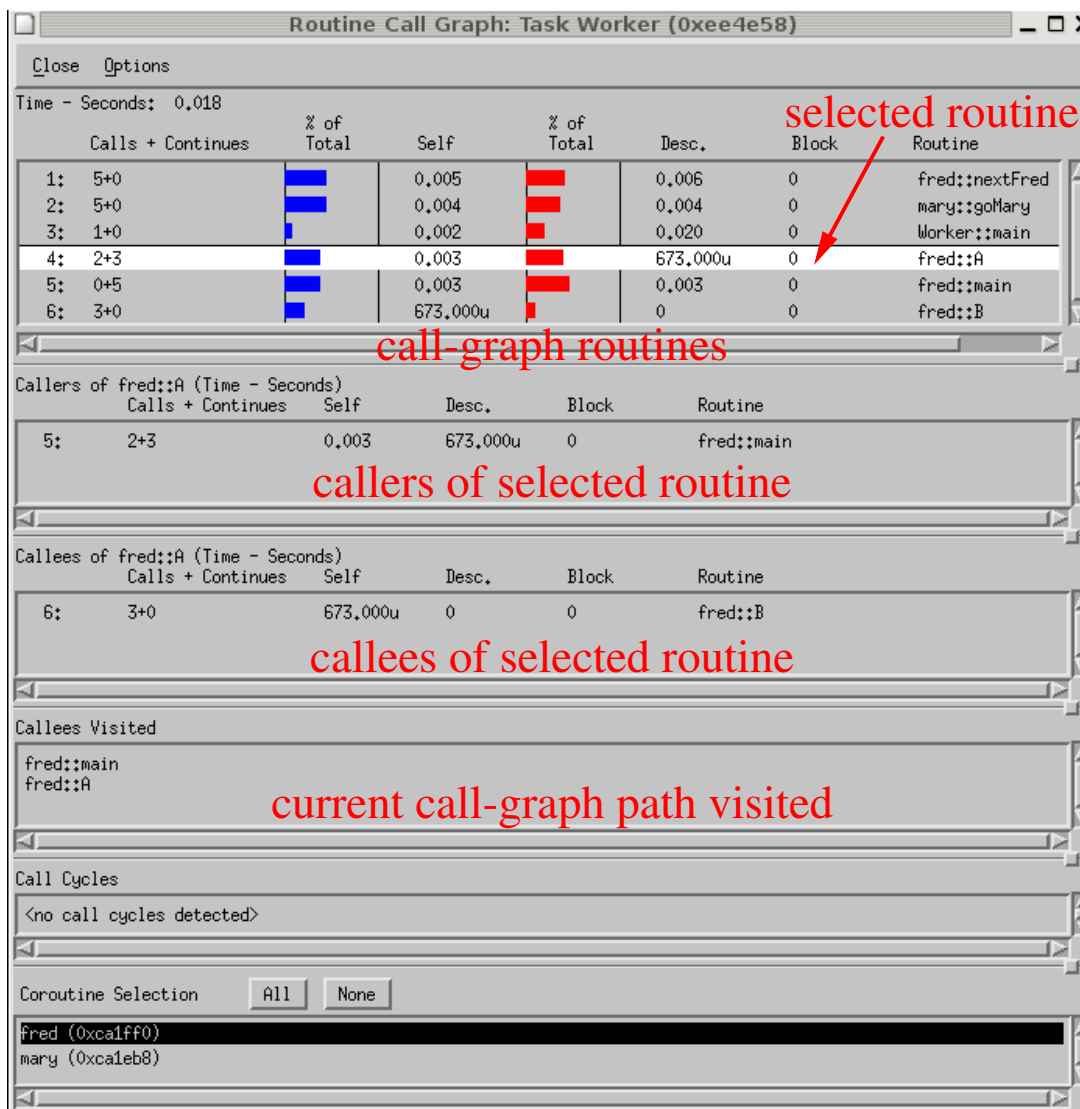
# Execution-State Transition Metric

- state transitions for each task
- scales to long duration and high magnification (up to 0.1 ns per pixel)
- increase/decrease magnification and adjust % change in magnification
- select task to display the list of transitions, including start/duration times and routine in which transition occurred



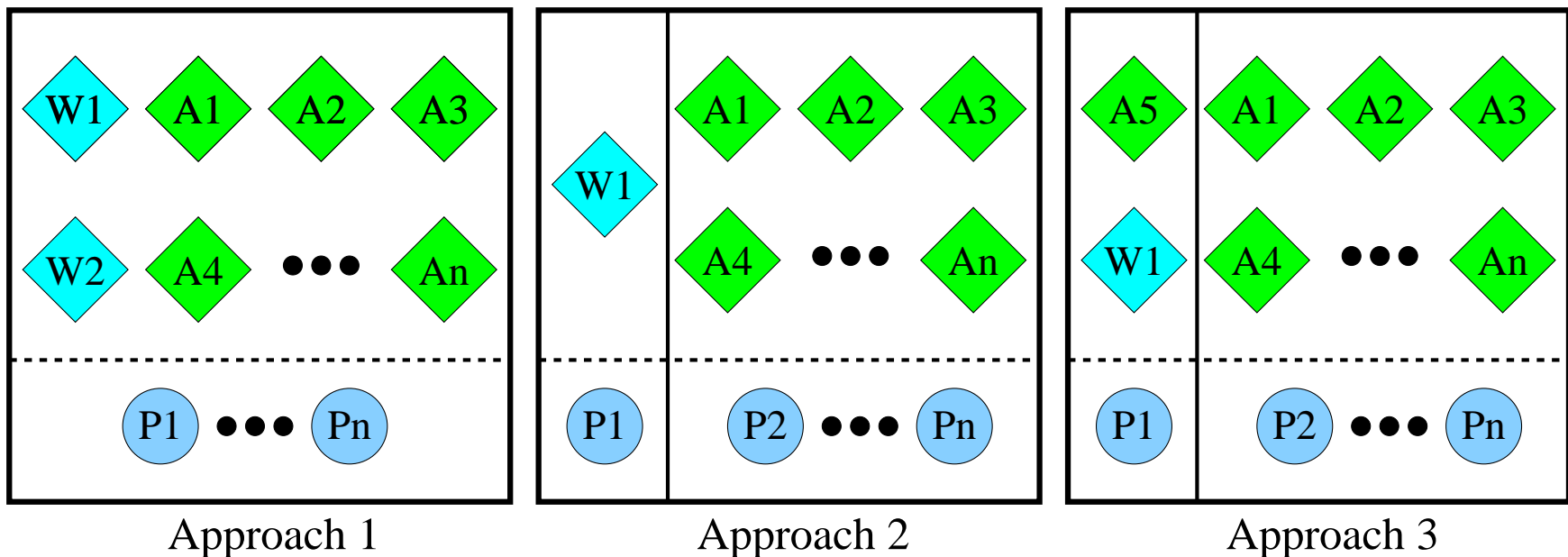
# Call Graph Metrics

- highly accurate exact or statistical call graph with sophisticated handling of call cycles
- measure hardware events or time
- select counting for user and/or system code
- statistical sampling frequency selectable for each event
- does not suffer from the gprof fallacy
- per task and coroutine call graphs
- selectively add/remove tasks and coroutines from call graph
- navigate call graph by selecting routines in caller and callee panes
- display complete gprof-style call graph (from Options menu)



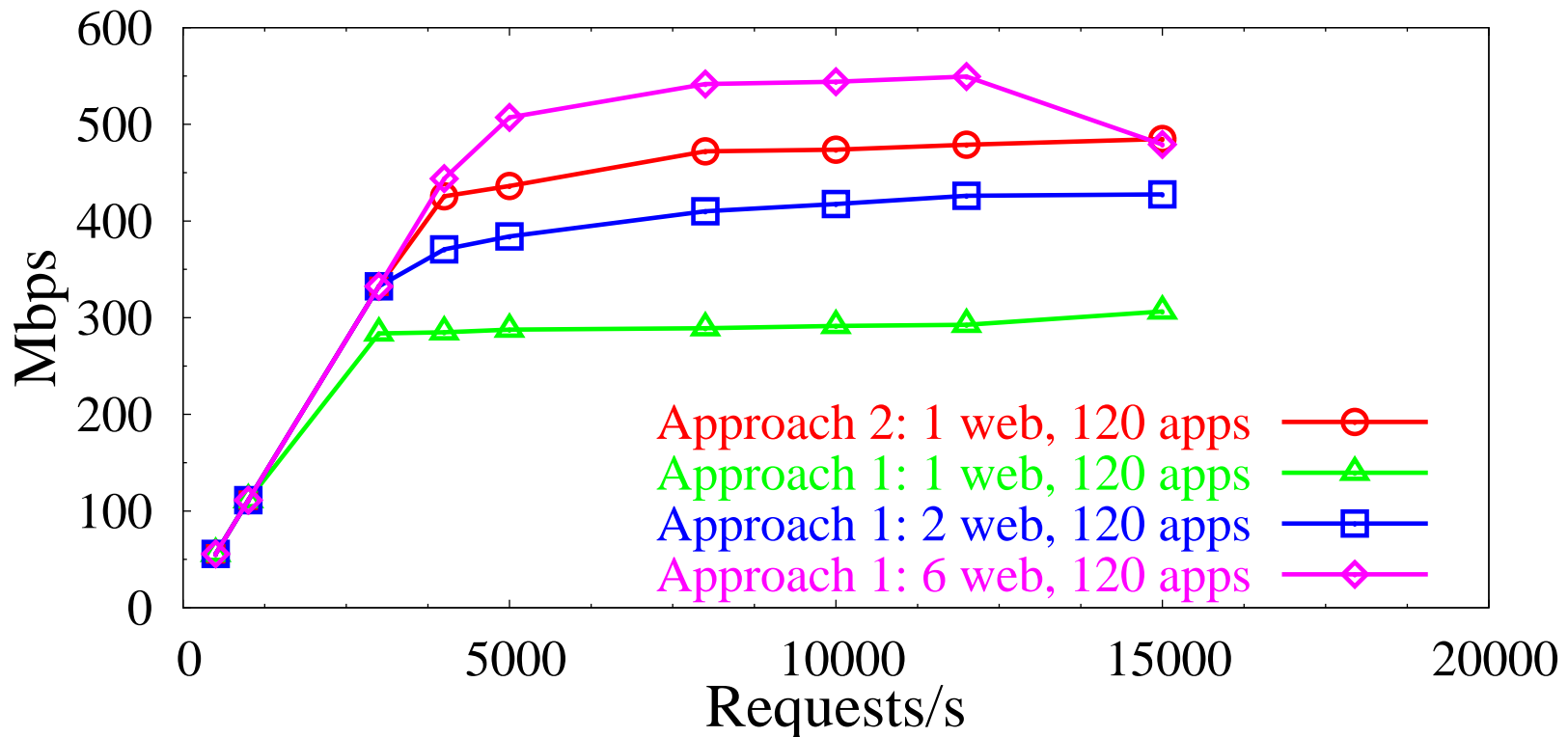
## Serving Dynamic Web Content on an SMP Computer

- Handling large volumes of client requests only requires a few web-server processes but many application processes.
- There are multiple ways to organize this structure on an SMP computer.



- Approach 1: allow operating system to schedule any processes on any CPU.
  - **Problem:** web-server processes (W1, W2) are overwhelmed by application server processes (A1..An) so web servers do not receive enough CPU.

- Approach 2: isolate web-server processes to a single CPU.
  - E.g., web server (W1) on CPU #1 and applications (A1..An) on CPUs #2-n
  - Ensures web server has more processing power.
  - **Problem:** idle cycles on web-server CPU reduce performance.



4 IA-64 CPUs, 4 GB Memory

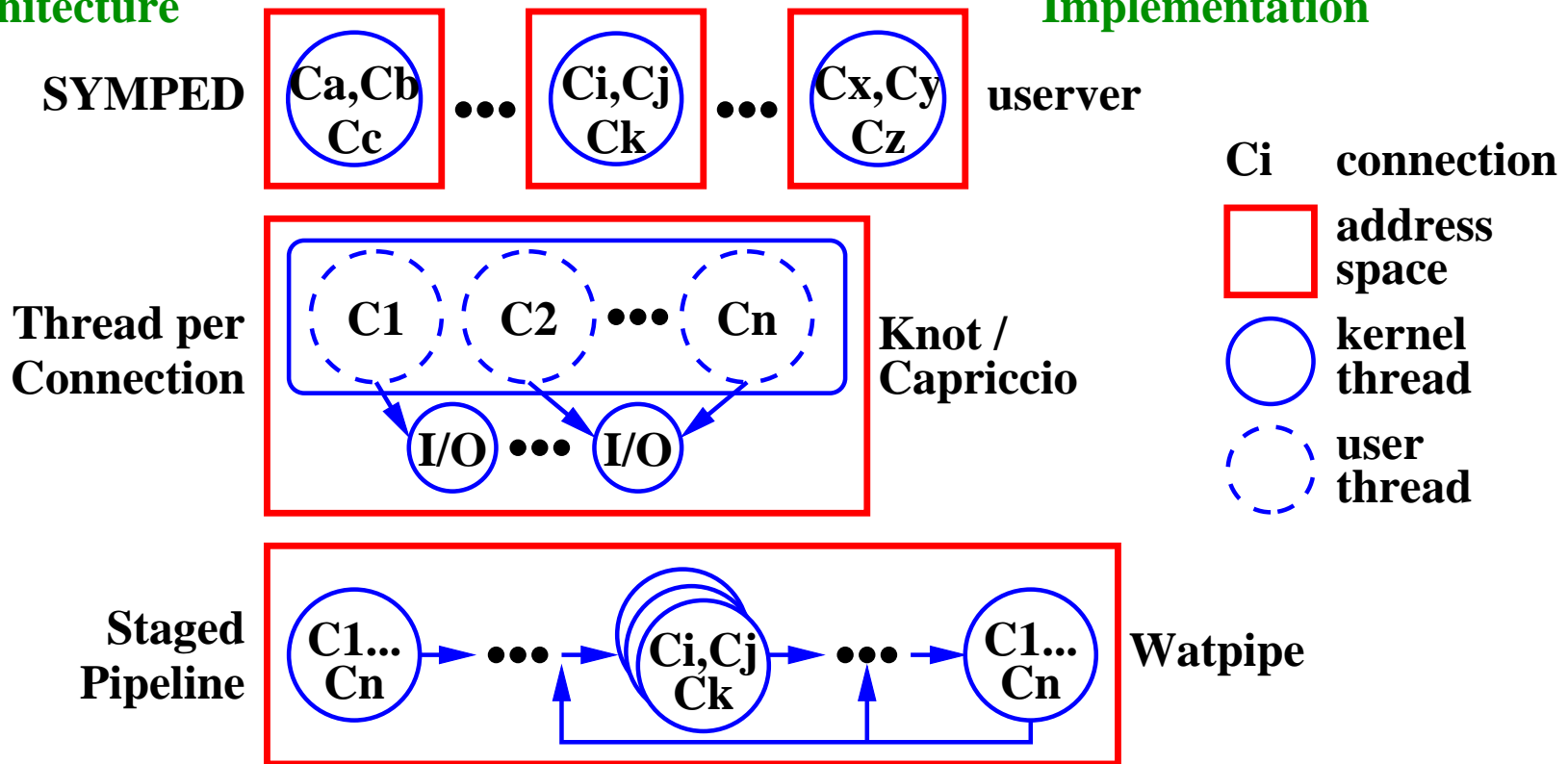
- Approach 3: move a few application servers to the web-server CPU.
  - Extra application servers could use idle cycles (future work).

# Comparing Web-Server Architectures

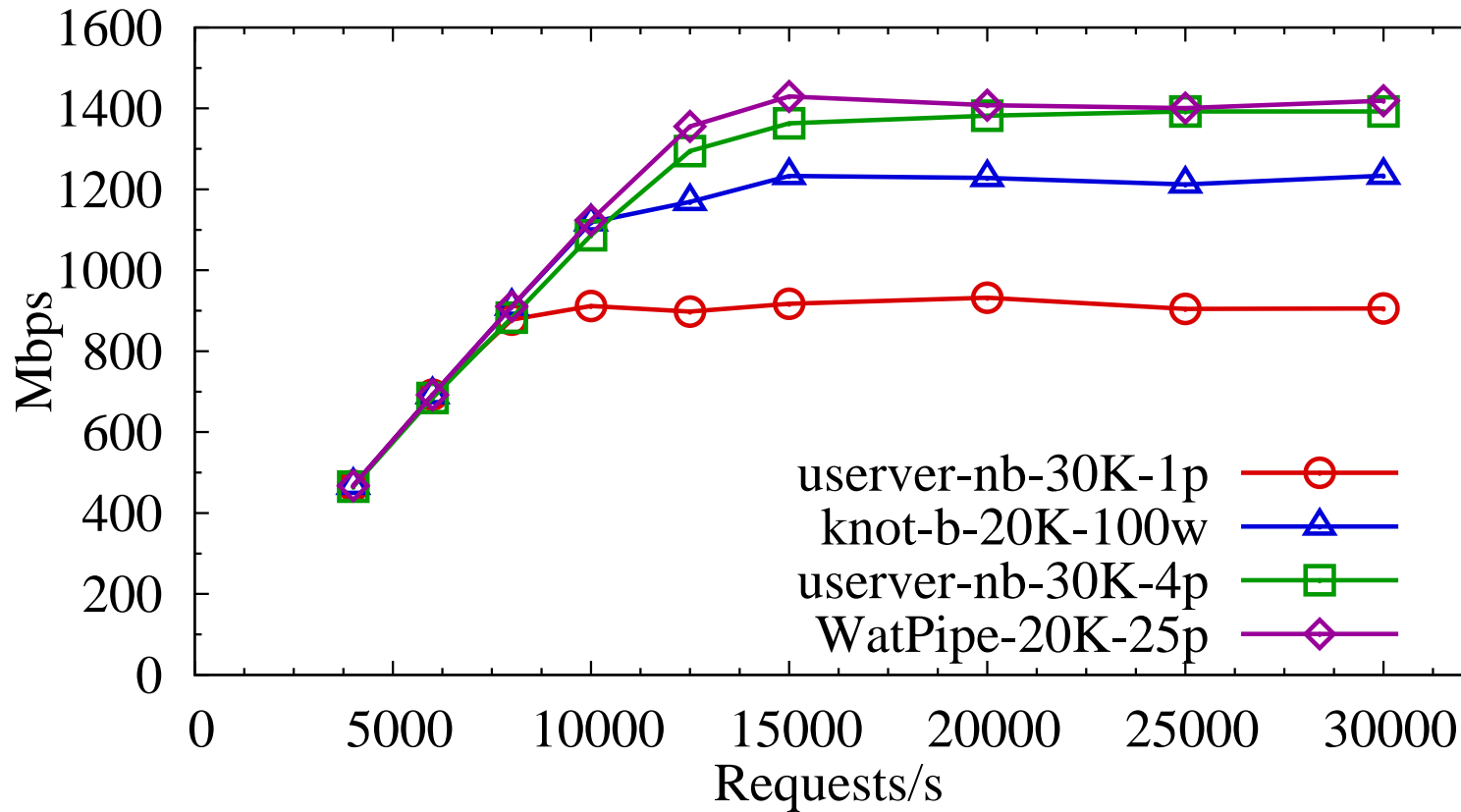
- Web server transfers data from disks (file content) through sockets to clients.
- These I/O operations may block awaiting completion, blocking server thread.
- Unix provides non-blocking I/O for sockets but only blocking I/O to files.
- To increase performance, servers perform non-blocking I/O and add threads.

## Architecture

## Implementation



## Performance of Server Architectures



- Non-threaded baseline with non-blocking sockets (red) does ok.
- Thread-per-connection (blue) has too many threads (overhead).
- Moderate number of threads works best (green/purple).
- *Comparing the Performance of Web Server Architectures*, David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla, EuroSys 2007, Lisbon, Portugal, March, 2007.