Reducing World-wide Web Latency

Venkata Padmanabhan

Aug 16, 1994

Outline

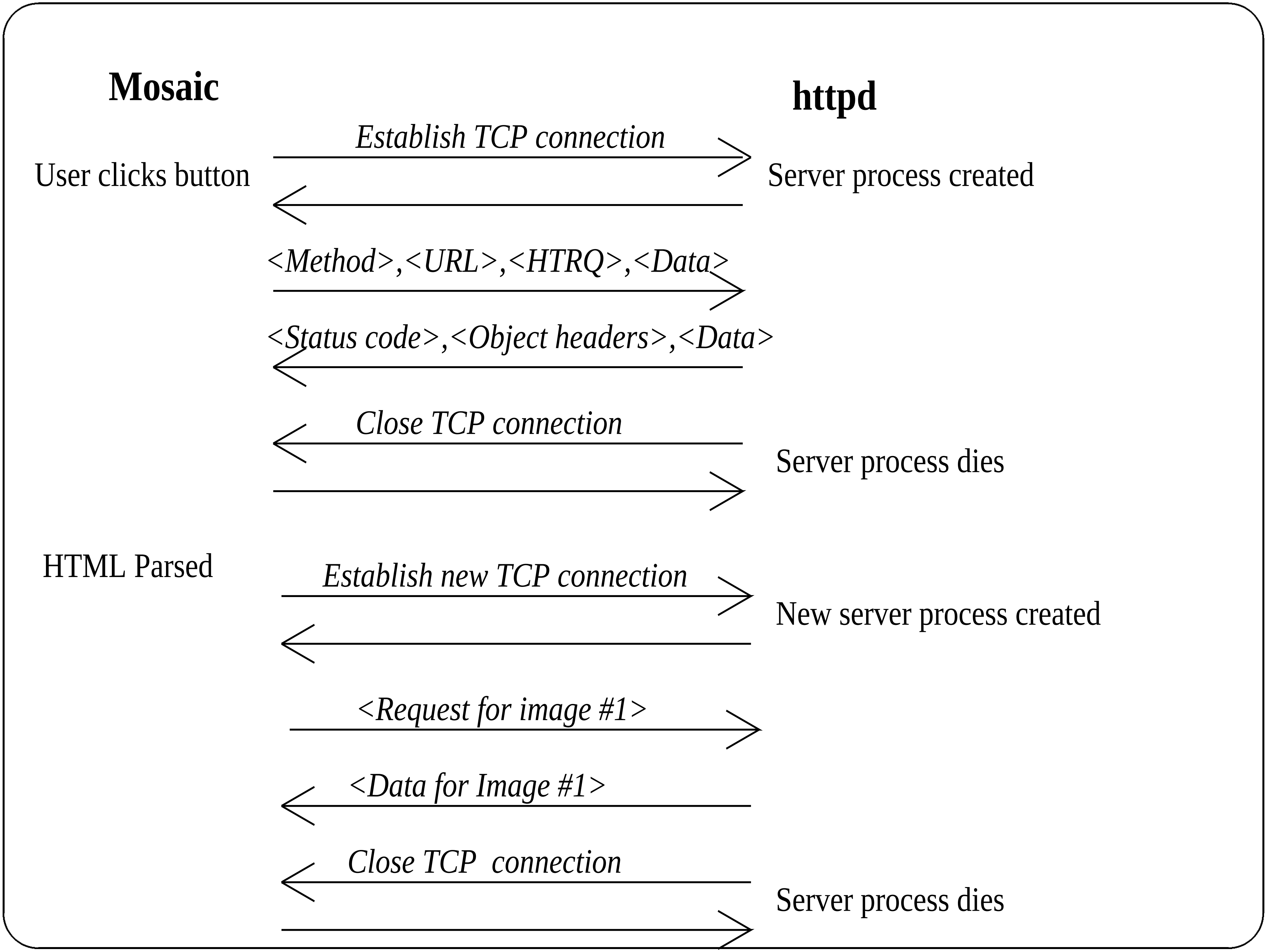
- Motivation
- Sources of Latency
- Mosaic - HTTP interaction
- Performance problems
- Modifications to the protocol
- Results
- Prefetching scheme
- Conclusions

Motivation

- The Web is slow at times
- **Main Reason:** The HTTP protocol is simple but inefficient

Sources of Latency

- Server: CPU and disk speeds
- Client: same as above
- Network:
  - bandwidth
  - Round-trip time (RTT)


Mosaic
httpd
Establish TCP connection
User clicks button
Server process created
<Method>,<URL>,<HTRQ>,<Data>
<Status code>,<Object headers>,<Data>
Close TCP connection
Server process dies
HTML Parsed
Establish new TCP connection
New server process created
<Request for image #1>
<Data for Image #1>
Close TCP connection
Server process dies

Problems

- Too many connections!
  - Processing overhead for each connection
    If authentication is done, that's extra overhead
  - 1 RTT each for set-up and tear-down alone
    (WRL-CRL RTT $\sim$ 75 ms for small packets)
  - TCP slowstart $\Rightarrow$ few connections reach full-steam

  On T1 line from WRL to CRL: sending 20000 bytes achieves a throughput of only 0.6Mbps
- No pipelining
  $\Rightarrow$ each inlined image requires additional roundtrip

Long-lived Connections

- Client tells server to keep connection open
  - uses HTRQ (HT Request) headers
  - Future implementations can define a `hold-connection` pragma
- Server process loops waiting for request
- Server can close connections to limit its load

Sounds great, but ...

- How does the client know when to stop reading?

Alternatives

- use a special EOT character
  - inefficient due to character stuffing
- have a separate *control* connection
  - unnecessary overhead in the common case
- use the *Content-Length* information
  - works for HTML files, images
  - Scripts are a problem

    So the server just closes the connection

We chose the last alternative.

Pipelining requests: GETALL

- GET $< HTML\_document >$

  $\Rightarrow$ Server sends back only the document

We define:

- GETALL $< HTML\_document >$

  Server sends back document and all inlined images
- can be implemented using HTRQ headers

But there's a problem:

- The client caches image data

GETLIST

So we define another primitive:

- GETLIST $< URL\_list >$

  $\Rightarrow$ Server sends back all the requested documents

Overall scheme

The client

- uses GETALL for the first access
- keeps cache of images URLs of recently accessed documents
- uses GETLIST for subsequent accesses to request only images required.

Results: CRL server

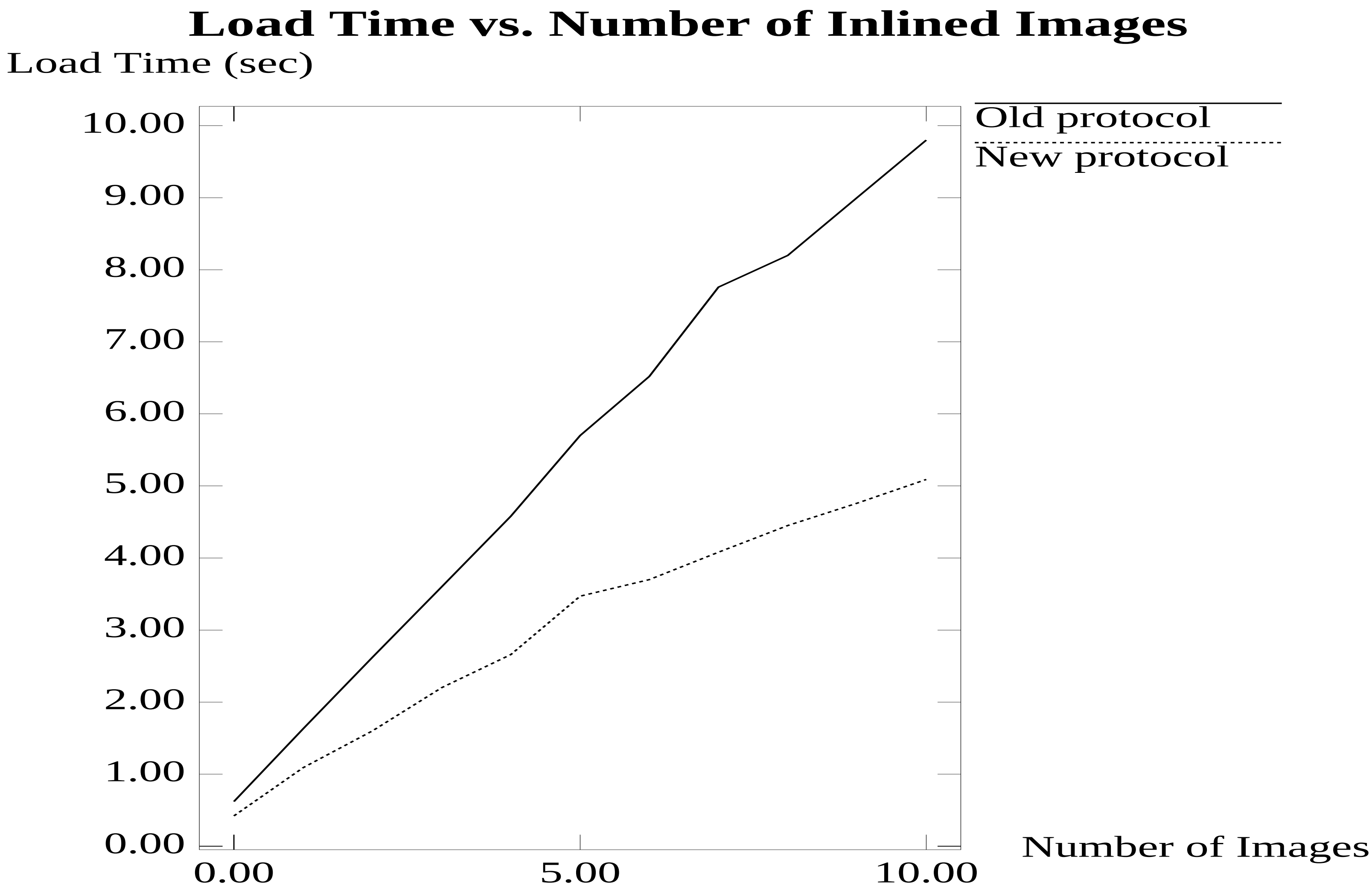


Image size: 2544 bytes

Results: CRL server

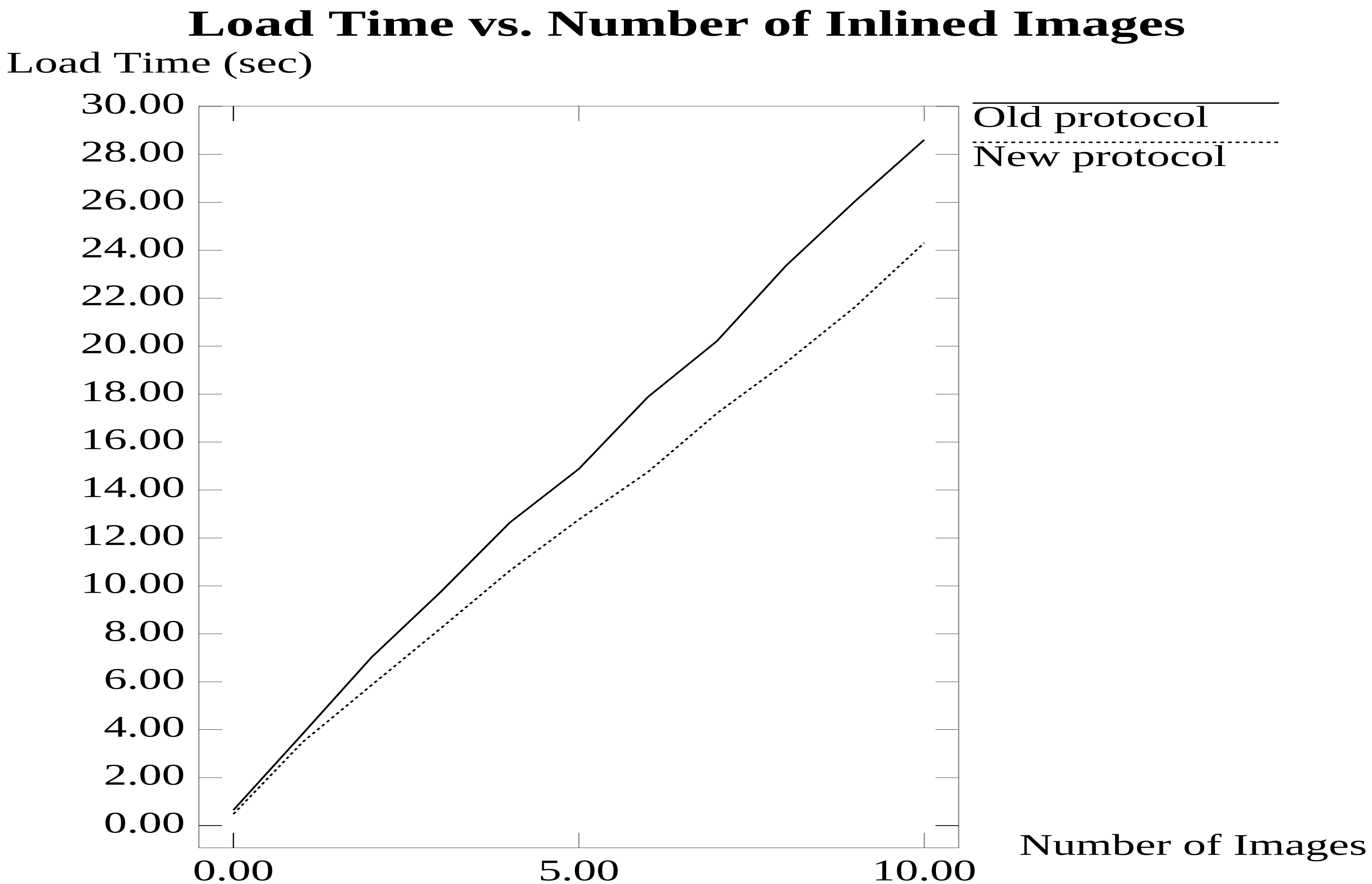
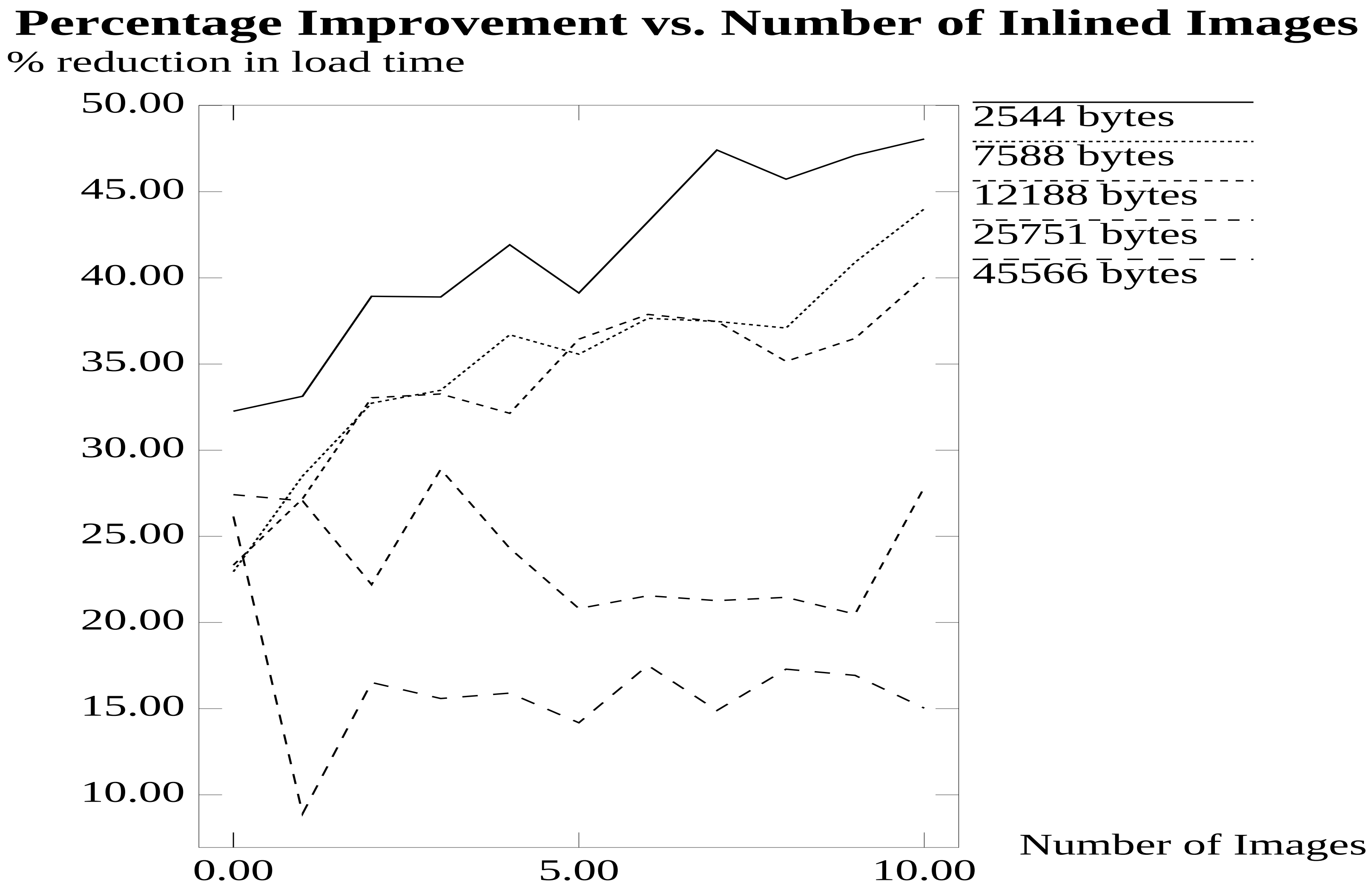
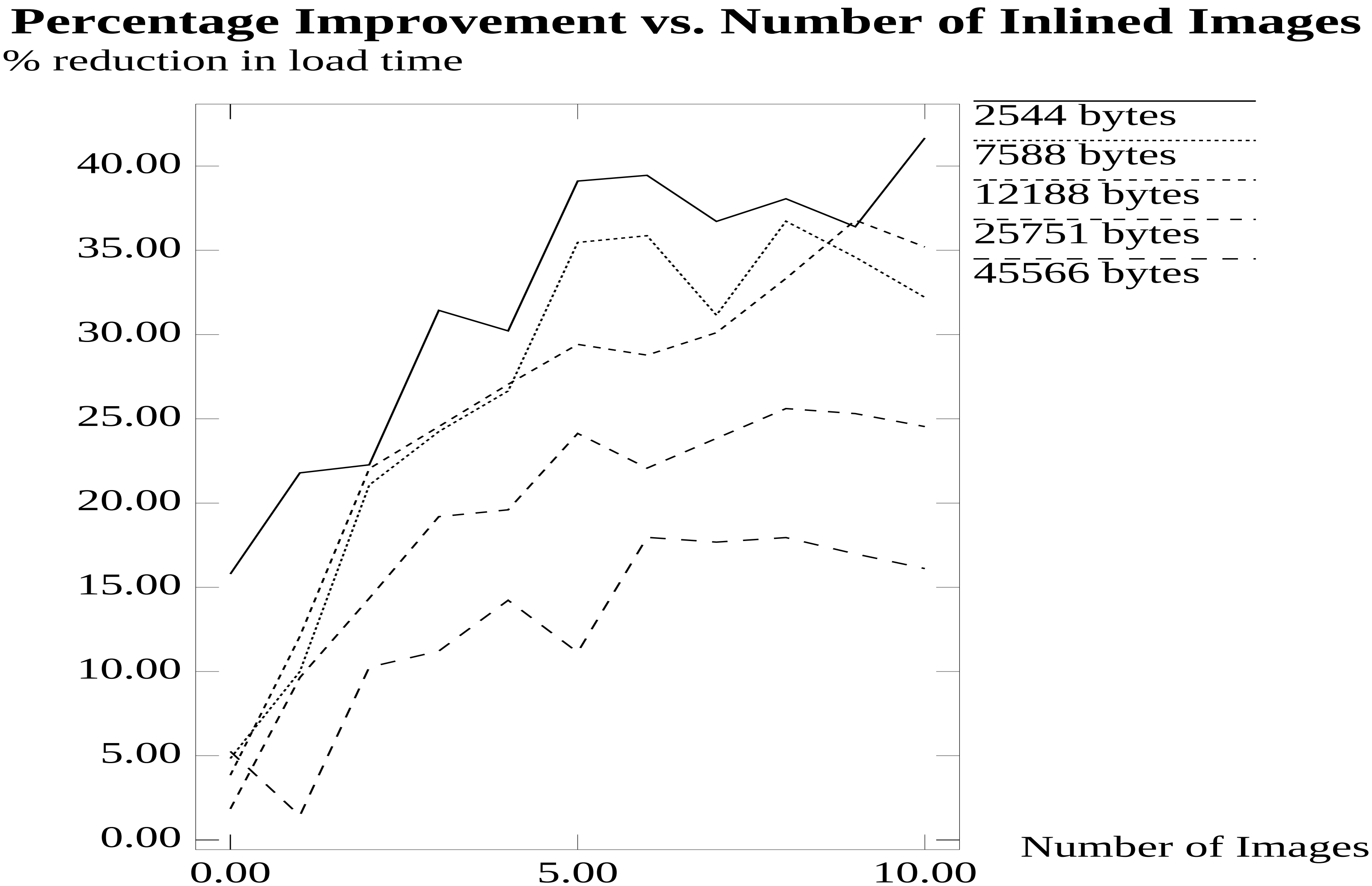


Image size: 45566 bytes

Summary: CRL server



Summary: WRL server



FTP Performance

Present Implementation:

- FTP Control connection re-established each time

  Problems:

  - increases latency
  - increases server load *(fork + exec)*
  - repeated authentication

Modification:

- hold connection open for a while
  - reduces latency

    Response time for browsing cut down to less than half
  - increases number of simultaneous connections for server

    But not worse than normal FTP

Prefetching by server

Basic idea: use past information to predict future requests

$\Rightarrow$ Prefetching can mask disk latencies

Issues:

- How to do it?
- Is it much use?
- Is server free enough?

How to do it?

Approach derived from Griffioen & Appleton [Summer USENIX '94]

Based on constructing a *dependency graph*

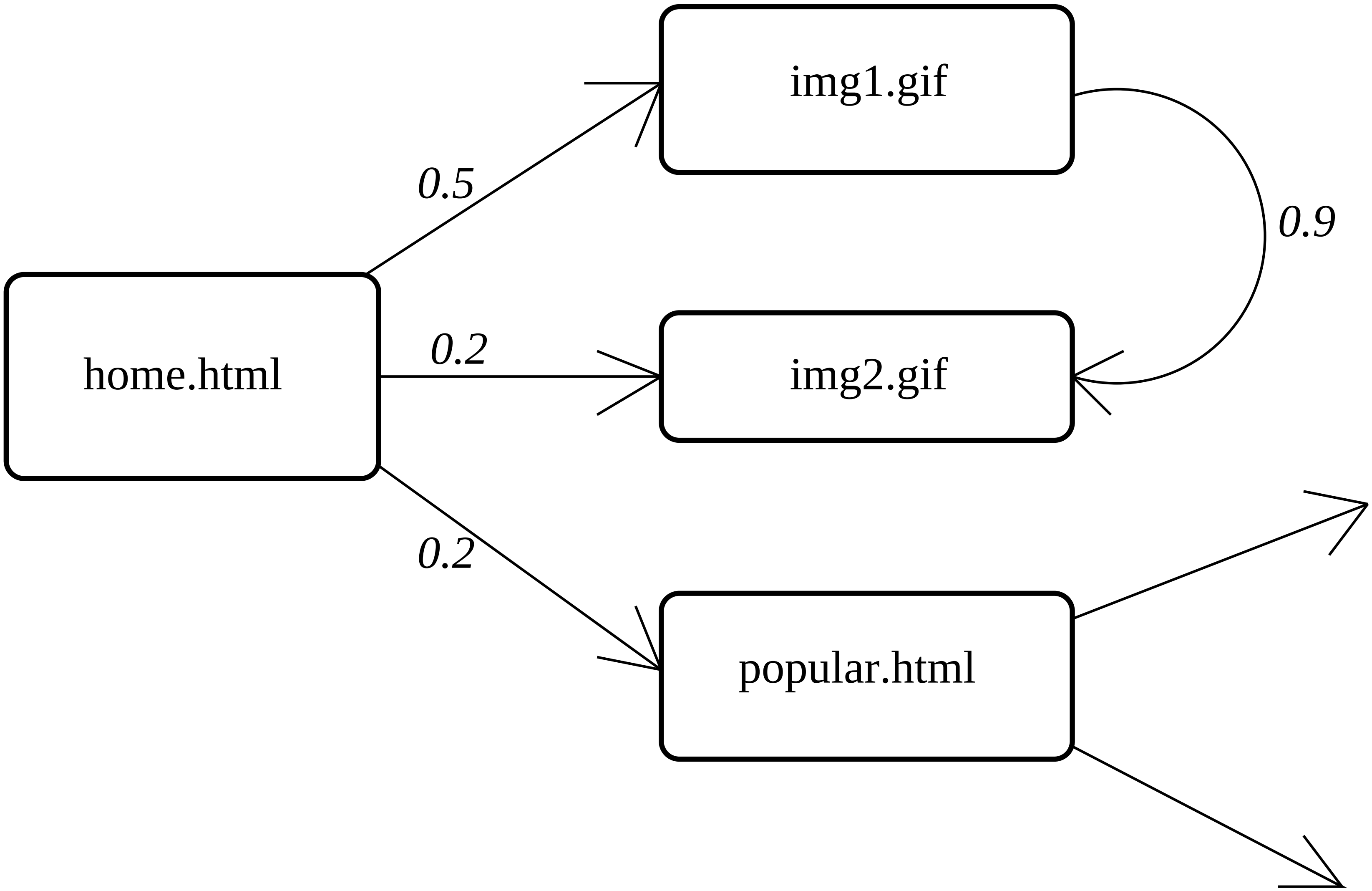
Parameters:

- lookahead window size (w)
- prefetch threshold (p)

Main differences:

- application driven
- maintain distinction between accesses by different clients

Dependency Graph



How much is to be gained?

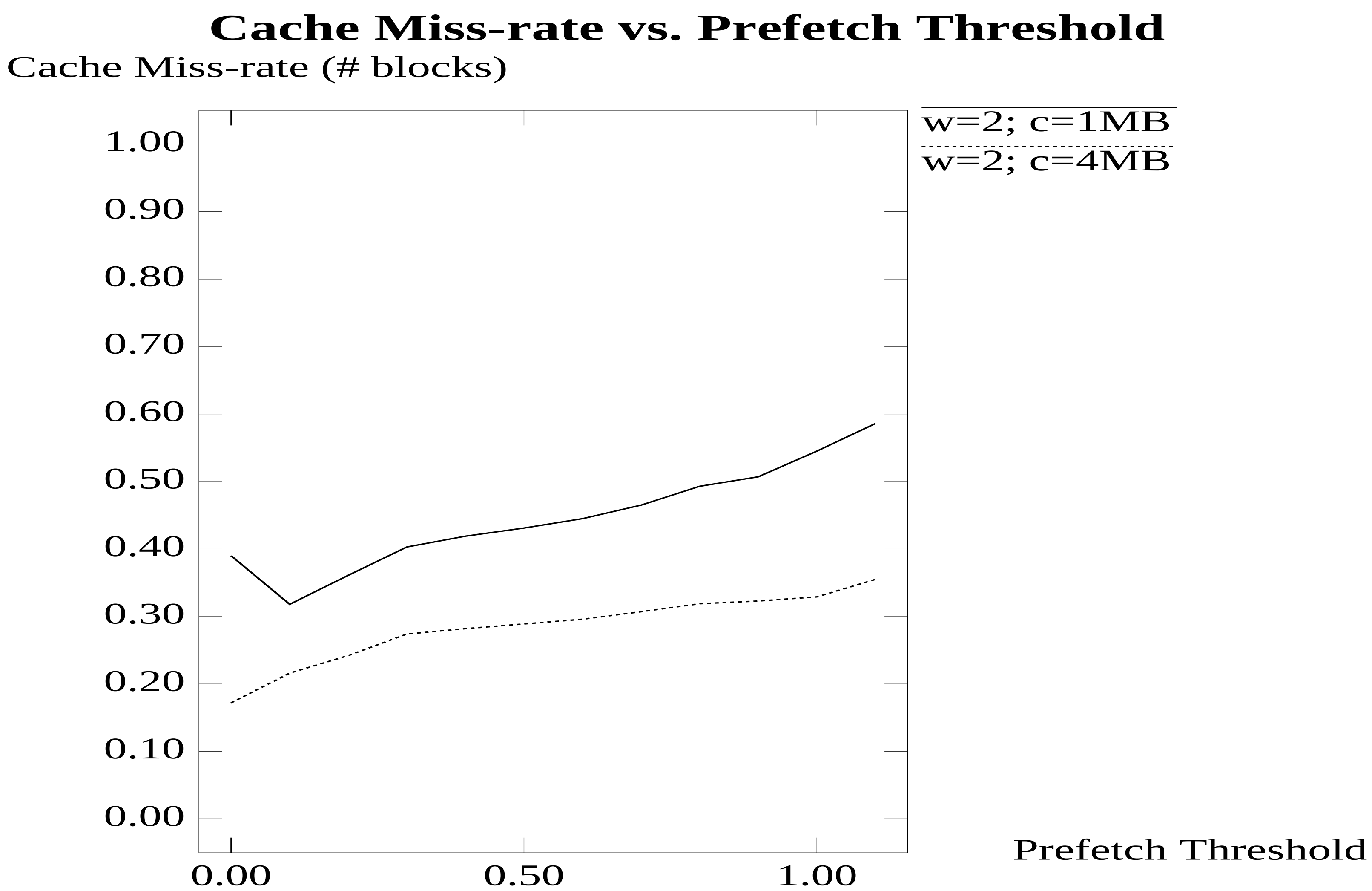
USENIX paper:

- studied filesystem accesses
- 30% arcs had estimated chance of 1
- upto 280% improvement in miss-rate
- individual accesses might take longer, but net performance gain

In our case:

- less dependency (only 6.5% arcs have estimated chance of 1)
  $\Rightarrow$ smaller improvement
- Is the server free enough?
  - not sure; need better traces
- will work better for local server

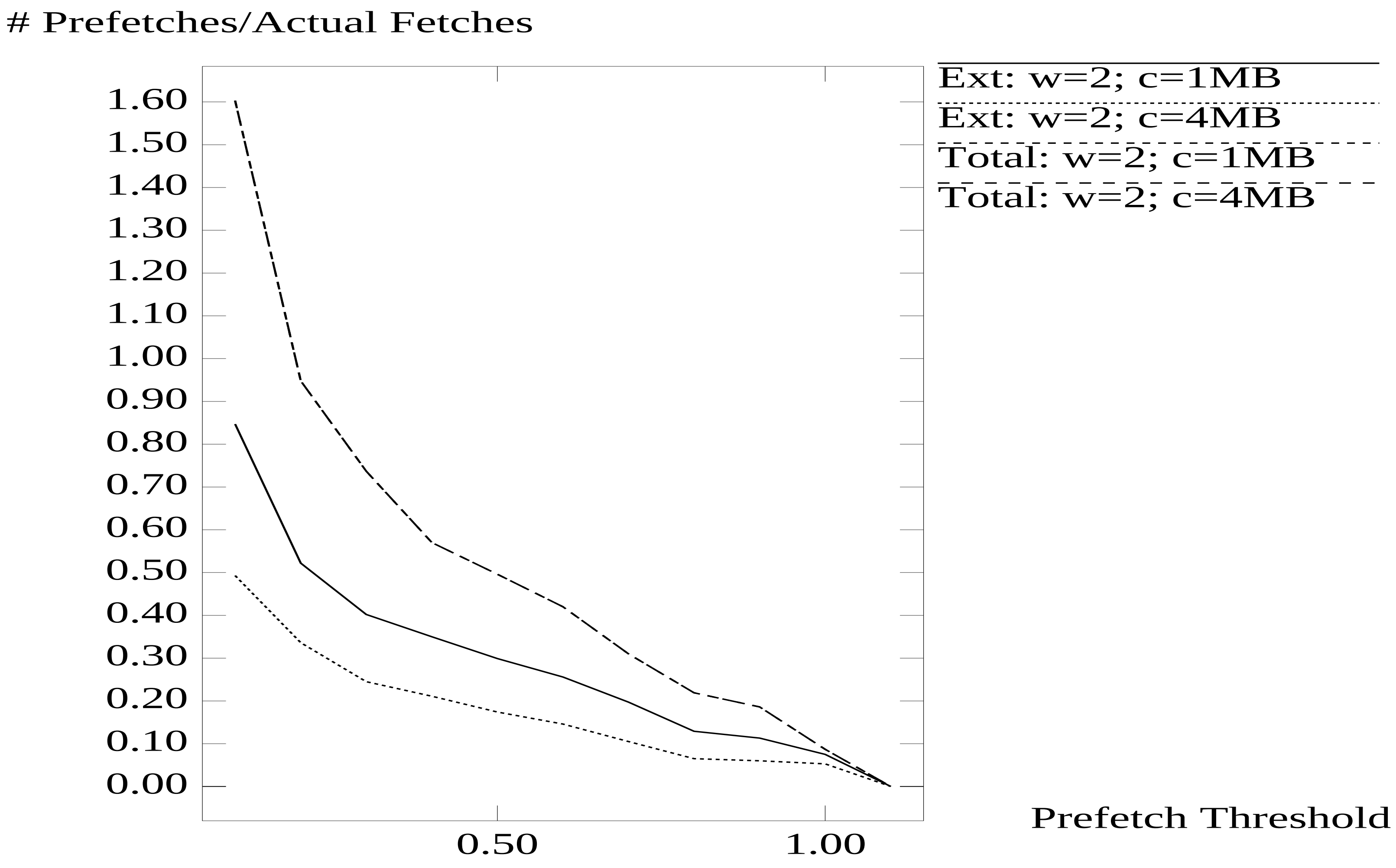
Simulation Results



Block size: 8 KB

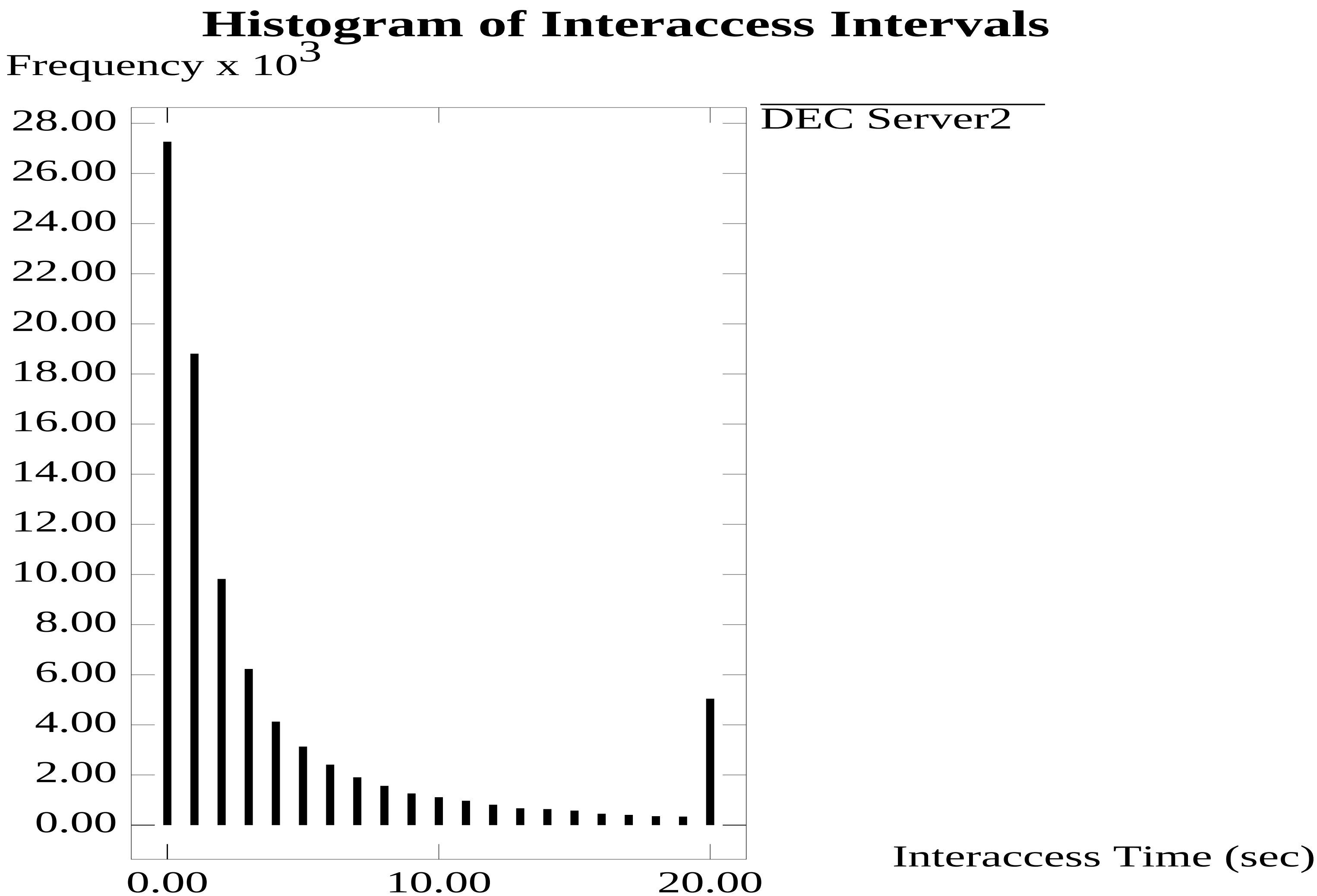
Simulation Results

Number of Prefetches vs. Prefetch Threshold



Block size: 8 KB

Simulation Results



Conclusions

- With a slightly modified protocol, there is a substantial reduction in latency
- Improvement depends on size and number of images
  - 15-50% for remote server
  - 10-40% for local server
- Full interoperability
- Basic problem of detecting EOT
- Prefetching might be useful

Future Work

- Complete study of server prefetching
- Investigate prefetching across the network