File Systems (Continued ...) and Database Issues in Mobile Computing

Jitu, Ming, Arvind and Jai

A Whole-File Prefetching Algorithm for Unix-like Environments

Motivation

- Burst loads on wireless links and mobile servers can be reduced by using prefetching.
- Prefetching also helps reduce latency experienced by clients.

Goal:

To develop a whole-file prefetching algorithm that is better than dumb algorithms incorporated in disk buffer managers but is still completely transparent to application level programs.

Three Prerequisites

- Excess network capacity between client and server
  - Required to absorb cost of “wasted” prefetches
- Excess CPU cycles on client and server
  - Required to make prefetch decisions (at client) and to serve them (at server).
- Suitable application structure

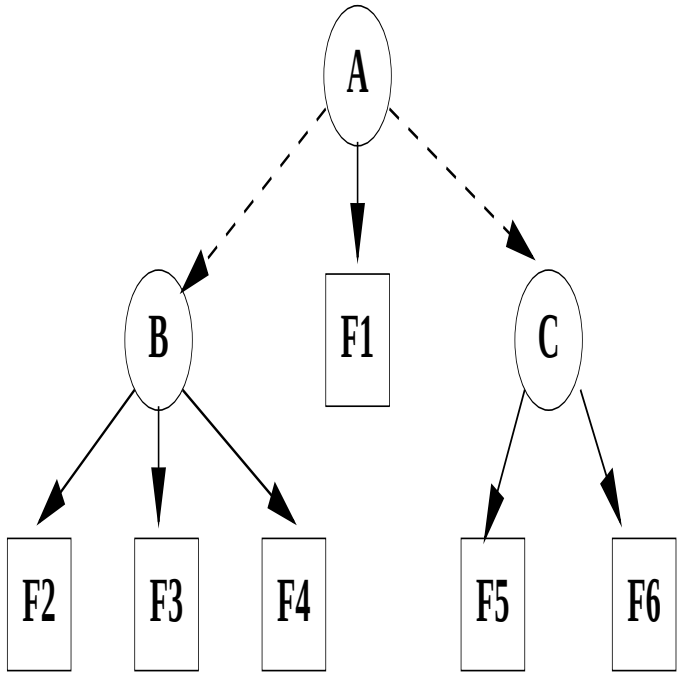
Intuition Behind the Algorithm

- History repeats itself!
  - Keep track of file accesses made by previous invocations of this application.
  - Use this trace data to make prefetch guesses
- Advantages of this approach
  - Better than “dumb” prefetching.
    - Influenced by application structure.
  - Completely transparent to applications
    - No need for application to inform prefetch daemon of its intentions

Requirements of Trace-Driven Prefetching

- An architecture to capture relevant system calls.
- Data structures for trace data
  - Easy to generate
  - Needs to contain all relevant information
  - Occupy as little space as possible
- A concrete metric of similarity between trace patterns.
  - Should be easy to compute
  - Robust

Application Trace in a Unix-like Environment



A Trace-Tree: Circles represent executables and Rectangles represent files.

- Circles are introduced in the trace by `fork` and `exec` calls.
- Rectangles are introduced in the trace by `open` and `chdir` calls.

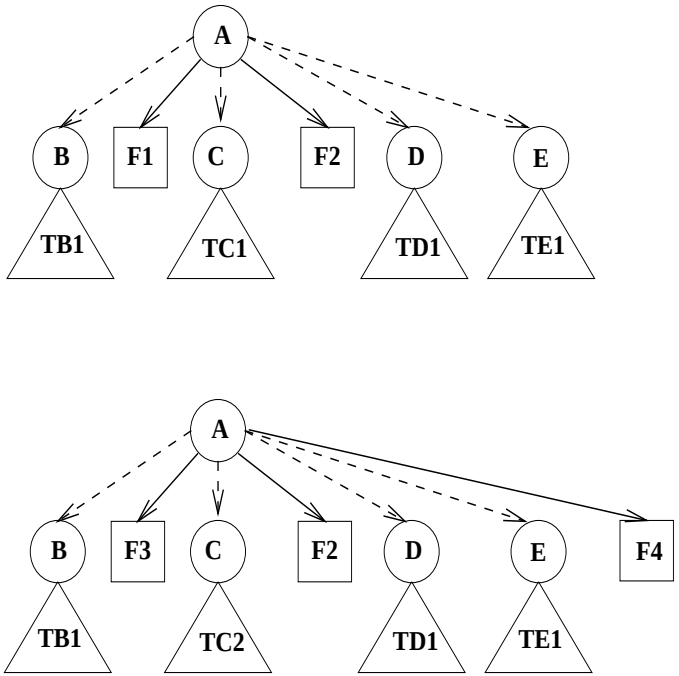
Two Simple Optimizations

- Ignore shell invocations
  - Data access patterns of command shells vary widely.
- Consolidate multiple consecutive accesses to the same file in one node
  - Most cache replacement algorithms will not move the file out of cache between multiple consecutive accesses.

Similarity Metric for Comparing Trace Trees

- Average of:
  - Percentage of data file accesses that match.
  - Percentage of executable files (fork and exec calls) that root identical access trees. (One-to-One correspondence required).

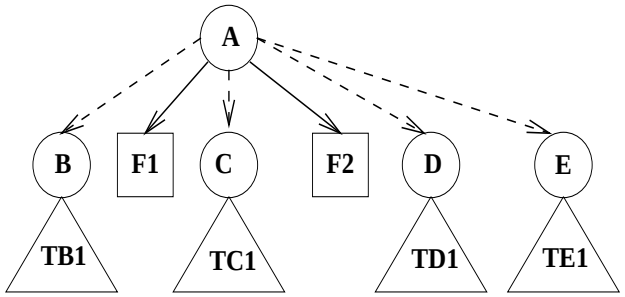
Example 1: Similarity between Complete Trace Tress

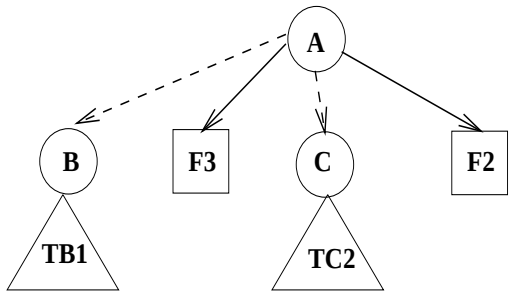


Two Complete Trace-Trees

- Matching files: 40%
- Matching executables: 75%
- Degree of match between the two tress: 57.5%

Example 2: Similarity between a Working Tree and a Trace Tree





- The pivot node is F4
- Matching files: 50%
- Matching executables: 50%
- Degree of match between the two tress: 50%

The Algorithm

- Trap each call to `fork, exec, open` and `chdir`
- Use this information to build an application tree
- As each node is added, compare the tree to previously stored trees
- If the value of the similarity metric exceeds pre-defined threshold, then prefetch the part of the tree if necessary.
- When execution is complete, store the finished tree if necessary.

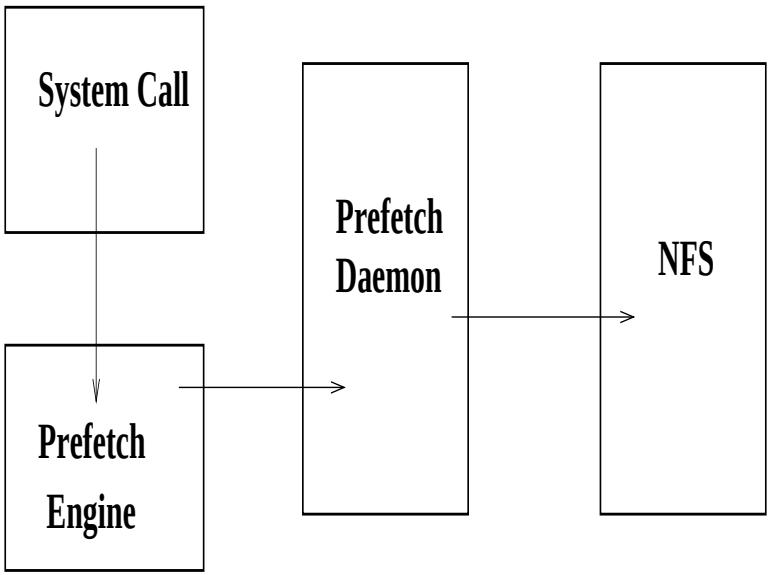
Performance Evaluation via Trace Driven Simulation

- Between 52%-92% improvement in cache miss rate when coupled with LRU replacement policy.
- Results are more impressive when cache sizes are smaller.
- Prediction accuracy is high:
  - Between 85%-96% of predicted files are actually used
  - The wasted bandwidth is less than 10% in most cases.
  - The compatibility function is robust (insensitive to small changes in threshold values)

Performance Evaluation via Implementation

Modified Unix server running on Mach operating system. Main observations:

- Latency over wired and wireless links is reduced by upto 67%.
- Even under antagonistic workloads, no increase under latency is observed.
- CPU overhead at client slide is negligible.



Conclusions

- Demonstrates how to use spare client CPU cycles to make better prefetching decisions. (Power consumption may go up).
- Completely transparent to application programs
- Successfully reduces latency

Related Work

- “Hoarding” in Coda file system
- TIP project requires applications to supply caching and prefetching hints
- Learning techniques used in Fido caching system for OODB

Selfish Questions:

Can some of these techniques be used to prefetch documents (text, images, audio and maybe even video) available on the web? Are user access patterns on WWW predictable in a general setting? Would the benefits be worth the trouble?