

CS-622

Advanced Computer Architecture Seminar

Syllabus highlights

- Instructor: Dr. Dmitry Ponomarev
 - Office: N-26 (for now, will move to the T-area soon)
 - Phone: (607) 777-4023
 - E-mail: dima@cs.binghamton.edu
- Course web page:
<http://www.cs.binghamton.edu/~dima/cs622>

Syllabus highlights

- Class meets on Tuesdays from 6 to 9 pm
- Class format:
 - Lectures
 - Students presentations of the recent conference papers
- Grading will be based on:
 - Class participation
 - Response papers
 - Final project

Response papers

- This will be assigned for all papers that we will read
- After you read the papers that are going to be discussed in class, write a 1-page document with your thoughts about these works and, hopefully, some brilliant new ideas of your own that were motivated by these works.
- Keep this document non-trivial, don't just describe strengths and weaknesses of the papers.
- You will only write 1 page per week, so put some thought into it!

Final projects

- Architecture students can just submit the status report on whatever they are currently working on as a final project.
 - But at least some initial results are expected!
- For other students, we will work on the topics together as the semester goes.
- At the end of the semester, projects will be presented in class.

Simulation Tools

- Tools: Can use SimpleScalar, or the various derivatives including those designed by our students.
 - Matt has a couple of simulators, Joe built SMT support on top of simpleScalar, etc.
 - Watch, Cacti for power and delay analysis
- Machines:
 - 11 dual-cpu linux machines in the "players" cluster (only dedicated to architecture research)
 - Generals
 - Matt is building a cluster of dual-core Athlon boxes

Computer Architecture in the Past

- For the last decade or so, researchers mostly focused on super-speculative architectures to boost the performance of superscalars
 - Branch prediction
 - Value prediction
 - Address prediction
 - Memory-dependence prediction etc.

What now?

- However, designs reached a performance plateau, where achieving additional gains becomes just too expensive
 - Traditional performance-oriented research is on a life support
- So? What are we supposed to do?

Computer Architecture: the New Frontiers (topics for this course)

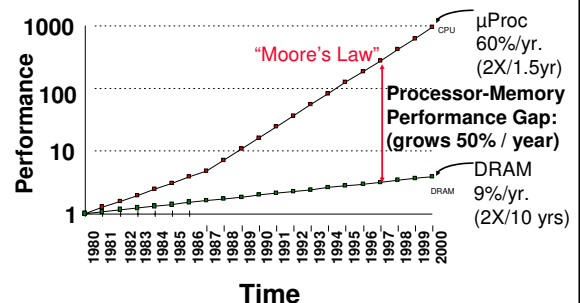
- Power-aware architectures
- Reliability-aware architectures
- Architectural support for security
- Support for software debugging
- Chip Multi-processing (CMP)
- Multithreaded Architectures (SMT, etc)
- Dynamic Optimization Systems
- Phase-based optimizations

However, to Get Started...

- There are two more traditional topics – data cache prefetching and branch prediction – that we will start with.
- These are always important, as the memory latencies continue to increase and other advances in speculation make the branch misprediction penalties ever more significant.

Data Cache Prefetching Techniques

Processor – Memory Gap



Data prefetching: motivation

- Due to the growing CPU-memory gap, a cache miss is becoming more expensive in terms of the number of CPU cycles
- **Data cache prefetching** is a technique that hides memory latency by bringing data into the cache before the processor actually requests it.

Data Prefetching Techniques

- Software prefetching
 - Special prefetch instructions are inserted into the program
 - This insertion is typically performed statically, by the compiler
- Hardware prefetching
 - Initiated dynamically by observing the program's behavior
 - Typically done at runtime by the hardware

Metrics of Prefetch Effectiveness

- Coverage
 - Fraction of original cache misses that are prefetched
- Accuracy
 - Fraction of prefetches that are useful
 - Useless prefetches waste memory bandwidth and can also displace useful data from the cache
- Timeliness
 - Determines how early the prefetches arrive
 - The full miss latency or only a part of it can be hidden
 - Too late or too early are both bad

Other Practical Considerations

- Cost and hardware complexity
- Whether code recompilation is required

Software Prefetching (SP)

- Relies on the programmer or the compiler to insert explicit prefetch instructions for memory references that are likely to miss into the cache
- SP historically has been effective for scientific programs that reference array-based data structures
- Recently, SP has been applied to non-numeric programs that reference pointer-based data structures

Hardware Support for SP

- The instruction set must provide a **prefetch instruction**
 - These are non-blocking memory loads which cause a cache fill on a miss, but otherwise have no effect
 - Can be ignored if the resources are scarce
 - Provide "hints" as to what should be loaded into the cache
- Requires lockup-free caches
 - Cache must continue servicing normal requests following the misses triggered by prefetch instructions
 - For aggressive prefetching, it is necessary to support multiple outstanding memory requests

Hardware Support for SP

- Support for ignoring memory faults caused by prefetch instructions
 - Especially useful when instrumenting prefetch instructions speculatively, for example to prefetch data accessed within conditional statements
- Possible use of prefetch buffers (PB)
 - Prefetched data is placed in PB
 - Data is moved from PB to the cache only on a hit
 - Delays displacing possibly useful data from the cache

Array Prefetching

- Array references performed within loops constitute regular memory access patterns
- Array subscripts are affine – i.e. linear combinations of loop index variables with constant coefficients and additive constants
- Quite common in a variety of applications:
 - Dense-matrix linear algebra
 - PDE solvers
 - Image processing etc.
- Memory access patterns can be identified at compile time

Mowry's Algorithm (ASPLOS'92)

- Involves three major steps:
 - Locality analysis
 - Cache miss isolation
 - Prefetch scheduling

Step 1: Locality Analysis

- Determines the array references that will miss into the cache and thus require prefetching
- Analysis proceeds in two parts:
 - First, reuse between dynamic instances of individual static array references is identified
 - Temporal, spatial and group reuses are considered
 - Determine which reuses result in cache hits
 - Compute the number of iterations and the amount of data accessed between reuses
 - If the size of this data is smaller than the cache size, then the reuse is assumed to hit and no prefetch is needed

Step 2: Cache Miss Isolation

- For static array references that experience a mix of cache hits and misses (as determined by locality analysis), code transformations are performed
- These isolate hits from misses in separate static array references

Code Transformations for Cache Miss Isolation

- Loop unrolling is used for spatial reuse
 - Example: Suppose that a loop performs a unit-stride traversal of array elements. Assuming 8-byte array elements and a 32-byte cache block, cache misses for each static array references occur every $32/8=4$ iterations. By unrolling the loop 4 times, leading array references incur cache misses every iteration while the remaining unrolled references never miss, thus isolating cache misses.

Code Transformations for Cache Miss Isolation

- Loop peeling is used for temporal reuse
 - Some subset of iterations (usually the first) incur all the cache misses, and the remaining iterations hit.
 - The cache-missing iteration(s) are peeled off and are placed in a separate loop.
- Nothing special is needed for group reuse, because cache misses already occur in separate static references.
- Once the isolation is complete, prefetches are inserted for the static array references that incur them.

Step 3: Prefetch Scheduling

- Due to high memory latencies, a single loop iteration typically has insufficient work to completely hide the cost of a cache miss
- To ensure timely delivery, prefetches must be initiated several iterations ahead of usage.
- Prefetch Distance (PD) is the minimum number of iterations needed to fully overlap a memory access
- By initiating prefetches PD iteration in advance, the first PD iterations are not prefetched while the last PD iterations prefetch past the end of each array.

Prefetch Scheduling

- These inefficiencies can be addressed by performing software pipelining to handle the first and last PD iterations separately.
- Software pipelining creates a *prologue loop* to execute PD prefetches for the first PD array elements, and an *epilogue loop* to handle the last PD iterations without prefetching.
- These are in addition to the original loop, which is called *steady state loop*.

Pointer Prefetching

- Pointer-chasing codes use linked data structures (LDS), such as linked lists, n-ary trees, and other graph structures that are dynamically allocated at run time.
 - Arise from solving complex problems, where the amount and the organization of data can't be determined at compile time
 - Also arise from high-level programming language constructs, such as those found in OOL

Pointer Prefetching

- Pointer-chasing codes commonly exhibit poor spatial and temporal locality, and experience poor cache behavior.
- Another bottleneck is the pointer-chasing problem.
 - Since individual nodes in the LDS are connected through pointers, access to additional parts of the data structure can not be completed until the pointer to the current portion of the data structure is resolved.
 - Poses problems for timely prefetches – address is not known for a while

Greedy Prefetching

- Simplest software prefetching technique for LDS traversal
- Proposed by Luk and Mowry (ASPLOS 1996)
- Inserts software prefetch instructions immediately prior to visiting a node for all possible successor nodes that may be encountered by the traversal

Greedy Prefetching for a Linked List Traversal

```
struct node (data, next)
  *ptr, *list_head

ptr = list_head;
while (ptr) {
  prefetch (ptr->next);
  .....
  ptr = ptr->next;
}
```

Greedy Prefetching for a Tree Traversal

```
struct node (data, left, right)
  *ptr;

void recurse (ptr) {
  prefetch (ptr->left);
  prefetch (ptr->right);
  .....
  recurse(ptr->left);
  recurse (ptr->right);
}
```

Data Linearization Prefetching

- Luk & Mowry (ASPLOS 1996)
- If contemporaneously traversed link nodes in an LDS are laid out linearly in memory, then prefetching a future node no longer requires traversing the intermediate nodes to determine its address.
- Instead, can simply offset from the current node pointer, much like indexing into an array.
- The key is achieving the desired linear layout of LDS nodes

Data Linearization Prefetching

- Linearization can occur either at LDS creation, or at runtime.
- Former is preferable, because no expensive data copying and LDS reorganization at runtime is required.
- Even then, periodic “re-linearization” is required if the nodes are added or removed frequently.
- Most effective for applications in which LDS connectivity does not change significantly during program execution.

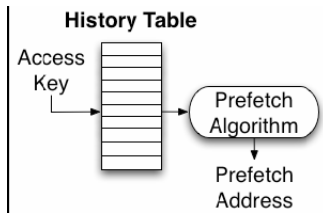
Jump Pointer Techniques

- Special pointers are inserted into the LDS solely for the purpose of prefetching
- These jump pointers connect non-consecutive link nodes, allowing prefetch instructions to name link nodes further down the pointer chain without traversing the intermediate link nodes and without performing linearization.
- Jump pointers increase the dimensionality and reduce the diameter of the LDS.

Hardware Prefetching

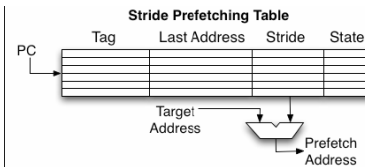
- Table-based prefetching
 - Stride prefetching
 - Correlation prefetching methods
 - Markov prefetching
 - Distance prefetching
- Prefetching using Global History Buffer
- Content-based prefetching
- Helper threads, Runahead execution

Table-based Prefetching



- Access key – load address, typically

Stride Prefetching



- This and all other figures are from “Data Prefetching Using a Global History Buffer” by K. Nesbit and J. Smith, HPCA 2004.
- Table is indexed by the PC of a load instruction

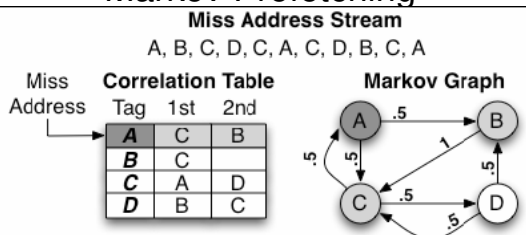
Stride Prefetching

- When a prefetch is triggered, addresses $a+s$, $a+2s$, ..., $a+ds$ are prefetched, where s is the stride, d is the prefetch degree and a is the current address of the load
- Less demands on the cache ports if implemented within L2 caches only.

Markov Prefetching

- Example of a correlation prefetching method
- Uses a history table to record consecutive address pairs
- When a miss occurs, the miss address indexes the correlation table
- Each entry in a Markov correlation table holds a list of addresses that immediately followed in the past the address that missed.
- All of these addresses are prefetched

Markov Prefetching

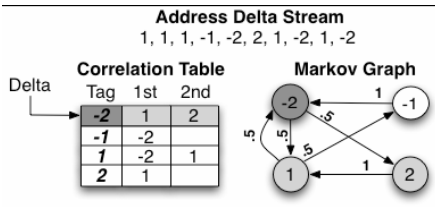


- Correlation table is a rough approximation of the graph at the right

Distance Prefetching

- Generalization of Markov prefetching
- Uses the distance between two consecutive global miss addresses – the address delta – to index into the correlation table
- Each correlation table entry holds a list of delta's that have followed the entry's delta in the past.
- One delta correlation can represent many miss address correlations, therefore this method is a generalization of Markov prefetching.

Distance Prefetching

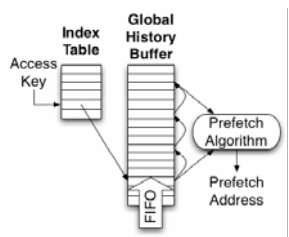


- To calculate prefetch addresses, the predicted deltas are added to the current miss address.

Inefficiencies of Prefetch Table Methods

- Table data can become stale, reducing prefetch accuracy
- Suffer from conflicts when multiple keys map to the same table entry
- Tables have fixed (and small) amount of history per entry

Prefetching using GHB

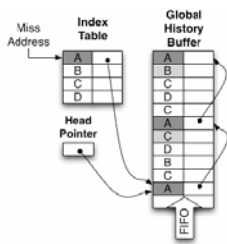


- Access key – load PC or cache miss address, or delta
- Entries in the index table point to the entries in GHB
- GHB holds n most recent L2 miss addresses

Global History Buffer (GHB)

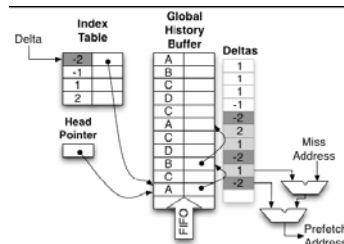
- A FIFO queue that contains N most recent addresses of the loads that missed into L2
- Each entry also stores a pointer
- The GHB entries are connected into link lists (using these pointers)
- Elements with the same table index key are linked together.
- Any number of history-based prefetch methods can be implemented based on the key used.

Example: Markov prefetching with GHB



- C and B are prefetched

Example: Delta Correlation Prefetching with GHB



Content-based Prefetching

- These technique examine the content of data being fetched to identify whether it contains an address or pointer that is likely to be accessed or dereferenced and prefetch it ahead of time
- Works for prefetching pointers in LDS

Content-based Prefetching

- Roth et.al.(ASPLOS 1998) proposed a technique to identify a pointer load, based on whether a load (consumer) is an address produced by another load (consumer). Proposed hardware that dynamically detects and prefetches various pointer loads.
- Cooksey et.al. (ASPLOS 2002) identified and prefetched pointers based on a set of heuristics applied to the content of a memory block

Content-Directed Prefetching (Cooksey et.al., ASPLOS 2002)

- When data is demand-fetched from memory, each address-sized word of the data is examined for a likely address.
- Candidate addresses need to be translated from the virtual to the physical address space and then issued as prefetch requests.
- As prefetch requests return data from memory, their contents are also examined to retrieve subsequent candidates.
- A set of heuristics are needed to identify "likely" candidates

Heuristics for Content-directed Prefetching

- Based on scanning each 4-byte chunk on every loaded cache line
- The first heuristics compares a few upper bits (called compare bits) of the candidate prefetch address with ones from the miss address.
- If they match, the candidate prefetch address may be an address that shares the base address bits with the miss address
 - This relies on the fact that LDS traversal may dereference many pointers that are located nearby.

Heuristics for Content-directed Prefetching

- Does not work well when upper most bits are all "0" or 1"s. Small integers or large negative numbers should not be confused with pointers!
- Ignoring these cases is unacceptable, because many Oses allocate heap and stack data in these locations.

Heuristics for Content-Directed Prefetching

- If compare bits are all 0's, next several bits (filter bits) are examined.
- If a non-zero bit is found in the filter bits, then the value is considered as a likely address.
- Finally, few least significant bits (align bits) can be useful at identifying pointers. The lower two bits of a pointer will be "00" if the addresses pointed to start at 4-byte granularity.
- Limitation: prefetch cannot be issued until the content of previous cache miss is available for examination

Other techniques

- Use helper threads
 - On an SMT, speculatively spawn a thread to pre-execute a piece of code that prefetches the data
 - By the time the load instruction is encountered by the main thread, the data will hopefully already be prefetched
- Runahead execution (Mutlu et.al., HPCA 2003)

Runahead Execution

- Normally, when an a load that missed into L2 cache blocks ROB, the pipeline stalls
- Runahead execution creates a checkpoint of the RF and unblocks the ROB
- While the cache miss is serviced, instruction down the line execute (and even pseudo-commit), generating more cache misses and effectively prefetching the data.