Assist Threads for Data Prefetching in IBM XL Compilers

Gennady Pekhimenko, Yaoging Gao, IBM Toronto, Zehra Sura, Tong Chen, John K. O’Brien, IBM Watson
Agenda

- Motivation and goals
- Compiler Infrastructure for Assist Threads
  - Delinquent load identification
  - Code region/Loop selection
  - Cloning
  - Back-slicing
  - Assist thread optimization: loop blocking, distance control
  - Speculation handling: version control and signal handling
- Performance Results
- Summary
Motivation and Goals

Motivation:
- Significant number of cache misses in HPC workloads
- Availability of multi-core and multi-threading for CMP (Chip MultiProcessors) and SMT (Simultaneous MultiThreading) exploitation
- Existing prefetch techniques have limitations:
  - hardware prefetch – irregular data access patterns
  - software prefetch – prefetch overhead on program execution

Goals:
- Deploy the available multiple SMT threads and cores to increase single thread and multiple thread performance
Compiler Infrastructure for Assist Threads

- Application programs
  - Delinquent Load Identification
    - Loop Selection
    - Region Cloning
    - Back-slicing
  - Basic AT Generation
- Profiling information for cache miss
  - Version Control
    - Distance Control
    - Loop Blocking
  - AT Optimizations

XL compiler

Binary with assist thread

AT Runtime

Thread management
Signal Handling
Assist Thread Generation

- Delinquent load identification (common to Memory Hierarchy Optimizations)
- Back-slicing for address computation and prefetch insertion
- Spawn assist threads once in MT (Main Thread)
- Notify AT (Assist Thread) to execute the slice function for delinquent loads

Main Thread

- Spawn assist thread
- signal assist thread
- Loop {
  ..........  
  delinquent load/loads
  ..........  
}

Inter-threaded communication

Assist Thread

- Slice function #id
  Loop {
    ..........  
    prefetch
    ..........  
  }
Example of AT Code Transformation

Original Code
// A and x are globals
y = func1();
...
i = func2();
// start of back-slice
while (i < condition) {
    ...
x = func3();
    ...
    // delinquent load
    func4( A[i] );
    ...
i += y;
}
// end of back-slice

Main Thread (MT) Code
y = func1();
...
i = func2();
// start pre-fetching in assist thread
func_addr = &slice_func_1;
signal assist thread;
// start of back-slice
while (i < condition) {
    ...
x = func3();
    ...
    // delinquent load
    func4( A[i] );
    ...
i += y;
}
// end of back-slice

Assist Thread (AT) Slice Function
void slice_func_1(int thd_id) {
    ...
    int y_local = y;
    int i_local = i;
    ...
    while (i_local < condition) {
        // pre-fetch request
        __dcbt( &A[i_local] );
        i_local += y_local;
    }
}
Delinquent Loop Identification

- Manual source change
  - Can be made by user using `__mem_delay()` function call
  ```c
  while (i < condition) {
    __mem_delay(&A[i]);
    A[i] += B[i];
    i += y;
  }
  ```

- Automatic identification using Performance Counters, -qpdf suboptions
  - Still `__mem_delay()`, but inserted automatically
  - Needs filtering for `__mem_delay()`, otherwise – too much overhead
Loop Selection

- Why do we need it?
  - Innermost, outermost or somewhere in the middle?
  - What if we have several delinquents at different levels?
  - Let’s try to avoid overhead

- What can we do?
  - Try to keep as many delinquents together as possible
  - Try to choose “hot” – enough loops, but without intermediate computation
  - Predict the amount to cache accesses based on the code in the loop
Back Slicing

- **Algorithm**
  - Start from the address expressions for all delinquent loads
  - Backward traversal of data and control dependence edges
    - Find all statements needed for address calculation
    - Remove un-needed statements from slice
  - Stores to global variables terminate traversal
    - Localization is applied when possible
  - Keep track of local live-ins to slice code
  - Insert prefetch instructions in slice
Controlling Progress of AT

- M(ain)T(hread), A(ssist)T(hread) execute asynchronously
- Prefetch may be useless, or harmful if AT-MT distance isn’t controlled
  - Too close => no latency hiding
  - Too far => potential cache pollution (and useless bus traffic)
- Reasons for the difference in pace
  - OS Scheduling of threads
  - SMT thread priorities
  - AT executes less code (a slice, rather than the real computation)
  - AT uses non-blocking prefetch
- Solutions
  - Version control (coarse grain)
  - Distance control (fine grain)
Version Control

- If AT lags, kill it when MT finishes loop
  - Prevents older AT prefetching for newer version of loop

- Avoid starting execution of stale AT
  - Increment version number on exit from every MT loop that has an associated AT
    - OS may schedule AT very late
    - AT compares current version number with version number at spawn point before starting execution

```c
signal assist thread (... , @version ....) 
Loop {
       ........
       delinquent load/loads
       .......
   } 
@version++; 

outlined_function(.... , version )
if( @version != version ) goto exit;
...
Loop {
       ........
       prefetch
       .......
} 
exit:
```
Distance Control

- Keep a reasonable iteration difference between MT, AT

Solution
- Introduce counters for MT and AT to record how many iterations have been executed
- If AT is too fast, wait
- If AT is too slow, jump ahead

Efficiency consideration in design
- All the checks are added to AT
- only increment of a counter is added to MT
  - Exact synchronization isn’t required
    - No locks needed
- Apply loop blocking to reduce the overhead
Loop Blocking

- Reduce overhead of distance control in MT
  - Block loop, increment counter only in outer loop
  - Number of increments decreased by blocking factor
  - AT distance control is less precise
    - Not that significant

```c
// MT Original Loop
for (i ; i < UB; i ++) {
    ... 
    a[i];
    ... 
    count++;
}
```

```c
// MT Blocked Loop
for (j; j < BF; j++){
    for (i ; i < min (BF*j + BF, UB); i ++) {
        ... 
        a[i];
        ... 
    }
}
count+=BF;
```
Speculation handling

- Speculative precomputation in AT may cause invalid accesses to global variables

- Static analysis to avoid speculation
  - But it is not always possible

- Runtime handling of speculation
  - Catch signal, skip this instance of AT, and continue execution
  - Both on Linux and AIX
Kernels Description

- Synthetic test cases to show the performance of assist thread on:
  - Different function unit usage
  - Different cache miss rate

- Operations in main thread can be grouped into:
  - ADDR: operations needed by AT to calculate the addresses for prefetch
  - COMP: the rest operations (computation) done by MT but not AT

- Different ratio between ADDR and COMP
  - at-comp: much more operations in COMP
  - at-addr: much more operations in ADDR
  - at-bal: ADDR and COMP are roughly balanced

- Different cache miss rate for delinquent loads ONLY
  - High: miss rate: ~90%
  - Medium: miss rate: ~40%
  - Low: miss rate: ~20%
CMP Assist Thread Performance on Power5

Speedup with CMP assist thread on P5

- AT with distance control
- AT without distance control

Kernels: high, medium, low
Speedup of Benchmarks

Speedup with CMP Assist Thread on Power5

- art
- bzip2-kernel
- Ibm
- mcf
Summary

- The Compiler infrastructure has been designed and implemented
  - Delinquent load infrastructure integration
  - Profitability analysis for code region selection
  - Outlining and backward slicing
  - Assist thread optimizations: distance control, loop blocking
  - Speculation handling

- Performance gains have been demonstrated on a set of kernels
  - Small kernels with different types of workloads
  - Several SPEC2000/2006 benchmarks
Future Plans

- Heuristic to choose the proper “delinquent” targets, based on
  - Profile information
  - Cache miss rates
  - Loop structure

- SMT priority control for assisted thread

- Automatic binding based on the current architecture and topology
Acknowledgements

- IBM Toronto compiler development team:
  Gennady Pekhimenko, Yaoqing Gao, Khaled Mohammed, Raul Silvera, Roch Archambault, Sandra McLaughlin

- IBM Watson compiler research team:
  Kevin K O'Brien, Tong Chen, Zehra Sura, Kathryn O'Brien