Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads

Md Kamruzzaman    Steven Swanson    Dean Tullsen
Modern Architectures

Intel Core i7

AMD Phenom

IBM Power7

Performance does not always scale with core counts
Performance Scaling Problem

• Serial part becomes dominant as you scale more
  – Amdahl’s law

• Not all codes are parallelizable
  – Parallelization is hard

• Synchronization issues
  – Coherence, locks, barrier

How can we utilize so many cores?
Non-traditional Parallelism

- **Helper thread** assists the main thread computation
- Helper thread **prefetching** reduces average memory access latency

[Diagram showing shared cache with helper thread prefetching data for main thread]
Helper Thread Prefetching

Simultaneous Multithreaded (SMT) Processor

- Threads share execution resources
- Helper thread competes with the main thread for cache space and bandwidth
Helper Thread Prefetching

Chip Multiprocessor (CMP)

- Only helps at distant inclusive shared caches and cannot leverage low latency of private caches
Migrating Helper Thread Prefetching

- Avoids resource competition problem
- Takes the benefit of private (e.g., L1/L2) cache prefetching
Migrating Helper Thread Prefetching

- No need of shared cache and can apply on any shared memory system
Outline

• Motivation

• Inter-core Prefetching

• Results

• Conclusion
Inter-core Prefetching

• Main thread **migrates** from core to core following helper threads and use the prefetched data
  – Migration converts cache misses into local hits
  – Improves **performance** and **energy**

• **Software-only** (e.g., compiler based) technique

• We apply Inter-core prefetching on a set of loops to improve single thread execution
Inter-core Prefetching

- Execution proceeds **chunk by chunk**
  - Helper threads **prefetch** data for future chunks in **private caches**
Inter-core Prefetching

- Execution proceeds **chunk by chunk**
  - Helper threads **prefetch** data for future chunks in **private caches**

![Diagram of inter-core prefetching]

- Threads swap cores
- Prefetching chunk 3
- Data already in the cache
- Main execution chunk 2
Inter-core Prefetching

- Execution proceeds **chunk by chunk**
  - Helper threads **prefetch** data for future chunks in **private caches**
Inter-core Prefetching

- Execution proceeds **chunk by chunk**
  - Helper threads **prefetch** data for future chunks in **private caches**

  - **Core - 0**
  - **Core - 1**

- Helper threads lead the way for the main thread

- Threads swap cores

- Prefetching chunk 5

- Main execution chunk 4
Flow Control Problem

- Helper thread is faster than the main thread
  - Prefetched data gets evicted before used
- Main thread is faster than the helper thread
  - Stall the main thread
  - Migrate before prefetching is done

Main thread and helper threads need to be synchronized
Synchronizing Threads

- Threads synchronize at the chunk boundary
Synchronizing Threads

- Threads synchronize at the chunk boundary
- Appropriate **chunk size** is the key
  - Larger chunks can overflow the cache
  - Smaller chunks imply more migrations / synchronizations
Main Thread Runs Faster

- Main thread stalls to match the slow prefetch thread
- Multiple prefetch threads can completely eliminate main thread stalling
Eliminating Main Thread Stall

- Core - 0
- Core - 1
- Core - 2

chunk 1
chunk 2
chunk 3
Eliminating Main Thread Stall

Core-0

chunk 1

chunk 4

Core-1

chunk 2

Core-2

chunk 3
Eliminating Main Thread Stall

- Core - 0
- Core - 1
- Core - 2

- chunk 1
- chunk 2
- chunk 3

- chunk 4
- chunk 5
Eliminating Main Thread Stall

Here, with 2 prefetch threads main thread does not wait and runs faster
Inter-core Prefetching Implementation

• Software only technique

• Need to predict future memory references
  – A lot of loops have this property

• Implementation
  – Identifying the loops
  – Decompose the loops into chunks
  – Synchronizing the main thread and prefetch threads
  – Generating code for prefetch threads
Identifying the Loops

• Profile to identify memory intensive loops

```c
jacobi2D () {
    for(i=1;i<1000;i++)
        for(j=1;j<1000;j++)
            a[i][j]=( b[i][j-1] +
                      b[i][j+1]+b[i-1][j] +
                      b[i+1][j] )* c
}
```

Original code

Memory intensive
Decomposing the Loops into Chunks

- Memory footprint of a chunk equals to the chunk size

Original code

```c
jacobi2D ( ) {
    for(i=1;i<1000;i++)
        for(j=1;j<1000;j++)
            a[i][j]=( b[i][j-1] + b[i][j+1]+b[i-1][j] + b[i+1][j] )* c
}
```

Main thread code

```c
jacobi2D_icp ( ) {
    for(chunk=0;chunk<100;chunk++) {
        for(i=chunk*10;i<(chunk+1)*10;i++)
            for(j=1;j<1000;j++)
                a[i][j]=( b[i][j-1]+b[i][j+1] + b[i-1][j]+b[i+1][j] )* c
    }
}
```
Adding Thread Management Code

- Thread creation, synchronization, etc.
- Efficient user-space thread migration

Original code

```plaintext
jacobi2D ( ) {
    for(i=1;i<1000;i++)
        for(j=1;j<1000;j++)
            a[i][j] = ( b[i][j-1] + b[i][j+1] + b[i-1][j] + b[i+1][j] ) * c
}
```

Main thread code

```plaintext
jacobi2D_icp ( ) {
    start_inter_core_pref ( )
    for(chunk=0;chunk<100;chunk++) {
        wait_and_swap(k+1)
        for(i=chunk*10;i<(chunk+1)*10;i++)
            for(j=1;j<1000;j++)
                a[i][j] = ( b[i][j-1] + b[i][j+1] + b[i-1][j] + b[i+1][j] ) * c
    }
    end_inter_core_pref ( )
}
```
Generating Prefetch Thread Code

• Reduced version of the original code

```c
jacobi2D_icp () {
    start_inter_core_pref()
    for(chunk=0;chunk<100;chunk++) {
        wait_and_swap(k+1)
        for(i=chunk*10;i<(chunk+1)*10;i++)
            for(j=1;j<1000;j++)
                a[i][j]=( b[i][j-1]+b[i][j+1] + b[i-1][j]+b[i+1][j] )*c
    }
    end_inter_core_pref()
}
```

```c
jacobi2D_pslice () {
    while(continue_prefetching){
        t = wait_and_swap(0)
        for(i=t*10;i<(t+1)*10;i++)
            for(j=1;j<1000;j+=8)
                load a[i][j], b[i][j]
    }
}
```

Main thread code

Prefetch thread code
Results
Evaluation

- Memory intensive benchmarks from Spec2K, Spec2K6, NAS, and micro-benchmarks
- Evaluated on a variety of real machines

![Diagram of computer architecture with memory sizes and CPU counts for Core2Quad, Nehalem, and Opteron.]
Factors That Impact Inter-core Prefetching

• Application characteristics
  • Data access pattern
  • Computation/memory ratio
  • Data sharing among chunks

• Prefetching parameters
  • Chunk size
  • Number of prefetch threads
Nehalem – Varying Chunk size

![Bar chart showing cache line access times for different cache sizes and access methods.](chart)

Higher is better

Number of prefetch threads used: 1
Nehalem – Varying Chunk size

Number of prefetch threads used: 1
Nehalem – Varying No of Prefetch Threads

Optimal chunk size/thread count depends on the code
Optimal Combination of Chunk Size and # of Cores for Different Loads

Inter-core prefetching is not constrained by traditional parallelization limit (e.g., 2x improvements using 2 cores)
Micro-benchmark - Inter-chunk Data Sharing

- **Read Sharing**
  - Core2Quad-base
  - Core2Quad-icp

- **Write Sharing**
  - Core2Quad-base
  - Core2Quad-icp

![Graph showing cache line accesses per microsecond for Read Sharing and Write Sharing]
Micro-benchmark - Inter-chunk Data Sharing

**Read Sharing**

- **Core2Quad-base**
- **Core2Quad-icp**
- **Opteron-base**
- **Opteron-icp**

**Write Sharing**

- **Core2Quad-base**
- **Core2Quad-icp**
- **Opteron-base**
- **Opteron-icp**

Cache line accesses / μs

0 5 10 15 20 25 30 35 40 45 50

0 0.1 0.2 0.3 0.4 0.5

0 5 10 15 20 25 30 35 40 45 50

0 0.1 0.2 0.3 0.4 0.5
Micro-benchmark - Inter-chunk Data Sharing

Inclusive shared cache can tolerate higher degree of write sharing
Benchmarks – Best Combination of Chunk Size and # of Prefetch Threads

Core2Quad – 63%, Nehalem – 31% and Opteron – 41%
Benchmarks – Energy Consumption for Core2Quad

- On average more than 25% energy savings, could be as much as 50%
Conclusion

- **Migrations** enable helper thread prefetching across cores to target private (e.g., L1) caches
- Inter-core prefetching requires **no hardware support**
- **Optimal chunk size/thread count** depends on application characteristics
- Speedup could be as high as **6x using 2 cores**
- Saves as much as **50% energy**
Thank You

Questions?