Speculative Precomputation on Intel Architectures*

Steve Liao, Dongkeun Kim, Perry Wang, Xinmin Tian, Hong Wang, Gerolf Hoflehner, Dan Lavery, Milind Girkar, John Shen

shih-wei.liao@intel.com
September 27, 2003

PACT Tutorial on Architecture & Compiler Support for Speculative Precomputation

* Disclaimer: This research work done in MRL does not represent any future products.
Outline

Scope: Target data prefetching on Intel® architectures

- No branch precomputation etc.

Part I. Binary-level tool on research Itanium® processors

- Chaining Speculative Precomputation (SP):
  - Helps in-order Itanium processors
- Experiments on Simultaneous Multithreading (SMT) Itanium processors

Part II. Source-level tool on IA-32

- Helper threads on Processors with Hyper-Threading Technology (HT Technology)
- Constructing helper threads
- Experiments on Pentium® 4 processors with HT Technology
Scope: Data Prefetching Threads

- Improve data latency of single-threaded codes using multithreading:
  - Use additional thread to prefetch for the main thread
  - Use program itself as predictor, instead of address pattern predictor
Part I. Binary-Level tool on Itanium

- Software-based approach: (Cf. Dynamic SP)
- Modest hardware support = SMT with few changes
  - Extend Itanium processors to SMT
    - 4 thread contexts
    - 8 cycles to activate a thread
  - Use off-line profiling to identify prefetching opportunities
  - No special hardware for register copying from main to helper
    - Rely on software to find live-ins & generate copying code

- Key: Tool to construct effective helpers
- Efficient helper: essential for performance
Binary-Level Tool

Post-pass tool

Current Intel compiler

Post-pass control flow graph and IR

Delinquent load identification

Slicing, scheduling, trigger point identification

Helper-threaded binary generation

Adapted binary with triggers+slices
Basic SP: 1 helper thread does it all

\[
\begin{align*}
A: & \quad t = \text{arc} \\
B: & \quad u = \text{load}(t->\text{tail}) \\
C: & \quad \text{prefetch}(u->\text{potential}) \\
D: & \quad \text{arc} = t + \text{nr\_group} \\
E: & \quad \text{branch if arc} < K
\end{align*}
\]

p-slice code for 1 helper thread:

\[
\text{do}\{ \\
\quad t = \text{arc}; \\
\quad u = \text{load}(t->\text{tail}); \\
\quad \text{prefetch}(u->\text{potential}); \\
\quad \text{arc} = t + \text{nr\_group}; \\
\quad \} \text{ while (arc} < K); \\
\]

Microprocessor Research Labs
Chaining SP: Addressing In-Order Itanium

Construct a doacross prefetching loop: Key in finding a p-slice that yields enough prefetch distance

+: long-range prefetching

+: helper threads progressing without hurting main thread

Main thread

Speculative loop: 1\textsuperscript{st} iteration

Speculative loop: 2\textsuperscript{nd} iteration

Microprocessor Research Labs
Chaining SP? Itanium Can Keep Prefetching (Cf. Basic SP)

(a) Original

(b) Chaining

(c) Basic
Chaining SP: Construct doacross loop

- Delay-Minimization for Chaining SP is an NP-complete problem
- 2-phase algorithm:
  - Dependence-Graph partitioning using strongly connected components (SCC)
  - SCC-partitioning tightens cycles on the dependence graph!
- Scheduling an acyclic graph

Diagram:

- Critical sub-slice: E -> D -> A
- Non-critical sub-slice: E -> D
- B: load
- C: load
- E: spawn
- D: spawn...

*: dependence edge
Critical Slice in Doacross Loop

For in-order processors such as current Itanium, if the load (“B:” above) misses, the machine stalls at “C:”

Scheduling should push computation into non-critical sub-slices as much as possible. Achieved by:

- Delay minimization via SCC-partitioning
- Dependence reductions
In-order & Out-of-order (OOO) Research Itanium Processor

- Modest hardware support: SMT with slight changes
- Thread-spawning: use existing light-weight mis-speculation recovery mechanism at user-level. (chk)
- Live-in copy: use on-chip memory buffer for Register Stack Engine.
### Modeled Itanium Details

<table>
<thead>
<tr>
<th>threading</th>
<th>SMT processor with 4 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelining</td>
<td>In-order: 12-stage. OOO: 16-stage</td>
</tr>
<tr>
<td>fetch,issue/cycle</td>
<td>2 bundles from 1 thread or 1 bundle each from 2 threads</td>
</tr>
<tr>
<td>window</td>
<td>In-order: 16-bundle expansion queue/thread. OOO: 255-entry reorder buffer/thread. 18-entry reservation station</td>
</tr>
<tr>
<td>registers/thread</td>
<td>128 integer, 128 FP, 64 predicates, 128 control, 8 branch</td>
</tr>
</tbody>
</table>
| cache        | L1: 16KB I- & 16KB D-cache. 4-way. 2-cycle latency  
L2: 256KB. 4-way. 14-cycle latency  
L3: 3MB. 12-way. 30-cycle latency |
| Memory       | 230-cycle latency. TLB miss penalty: 30-cycle |

For research, use higher memory latencies than current Itanium 2 processors
Slice Characteristics for 7 Pointer-Intensive Programs

<table>
<thead>
<tr>
<th>benchmark</th>
<th>slices (#)</th>
<th>average size</th>
<th>average # live-in</th>
<th>rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>em3d</td>
<td>8</td>
<td>10.3</td>
<td>2.8</td>
<td>chaining</td>
</tr>
<tr>
<td>health</td>
<td>2</td>
<td>9.0</td>
<td>3.5</td>
<td>chaining</td>
</tr>
<tr>
<td>mst</td>
<td>4</td>
<td>28.3</td>
<td>4.8</td>
<td>chaining</td>
</tr>
<tr>
<td>treeadd.df</td>
<td>3</td>
<td>11.3</td>
<td>3.0</td>
<td>basic</td>
</tr>
<tr>
<td>treeadd.df</td>
<td>2</td>
<td>12.5</td>
<td>4.5</td>
<td>chaining</td>
</tr>
<tr>
<td>mcf</td>
<td>5</td>
<td>14.0</td>
<td>4.4</td>
<td>chaining</td>
</tr>
<tr>
<td>vpr</td>
<td>6</td>
<td>13.5</td>
<td>4.0</td>
<td>chaining</td>
</tr>
</tbody>
</table>

Several static slices cover delinquent loads.

Slices are not big. #live-ins are not many.

Chaining SP is profitable when:
- non-critical sub-slice is large, or
- trip count is large, or
- thread spawning overhead is small

Source: Liao, PLDI’02
Speedup on in-order & OOO models

- SSP: Our Software-based SP
- Baseline: In-order processor without SSP
- On in-order: SSP improves 87%.
- On OOO: SSP improves 5%

Source: Liao, PLDI’02
Cache Latency Reduction Analysis

- Long-range prefetching reduces L3 misses
- On OOO, SSP reduces L3 misses for all 7 programs, but only 3 programs achieve speedups using SSP
  - Reason: SSP increases L1 misses.
  - Need to apply SSP judiciously on OOO (OOO already covers L1 misses)

Source: Liao, PLDI’02
Summary of Part I

- Minimal hardware changes: Use Software Tool instead
- For 7 pointer-intensive programs, several static slices cover many delinquent loads.

Even with conservative HW, SSP achieves 87% speedup on in-order processor. But 5% speedup on OOO.

SSP & OOO need to be complementary to deliver performance: SSP targets long-range L3 misses without polluting L1.

Motivated by this work, we applied SP to Pentium 4 Processors with HT Technology

Part II of this talk
Source-level Tool (“AutoHelper”) on IA-32

Motivation for AutoHelper study on IA-32

Arrival of Pentium 4 Processors with HT Technology

Evaluate simulator-based ideas

If manually constructing helper thread’s code:

Error-prone

Not providing systematic study or insight on HT Technology
Part II Outline

- Helper thread on processor with HT Technology
  - Exploit the extra logical processor on a processor with HT Technology
  - Hide latency for single-threaded codes via memory-level parallelism
- AutoHelper: a tool designed to exploit the above
- Case Studies
- Summary
Intel Hyper-Threading Technology Architecture

- SMT: Executes two tasks simultaneously
- Two different applications
- Two threads of same application
- CPU maintains architecture state for two processors
- Two logical processors per physical processor

Source: Intel Technology Journal’02
## Hardware Management in Processors with HT Technology

<table>
<thead>
<tr>
<th>Type</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shared</strong></td>
<td>L1 D-Cache, L2 Cache, Trace Cache, Execution Units,</td>
</tr>
<tr>
<td></td>
<td>Microcode ROM, Instruction Fetch Logic, IA-32 Instruction</td>
</tr>
<tr>
<td></td>
<td>Decode, Global History Array, Allocator, DTLB Instruction</td>
</tr>
<tr>
<td></td>
<td>Scheduler, Uop Retirement Logic</td>
</tr>
<tr>
<td><strong>Replicated</strong></td>
<td>Per-CPU architecture state (Instruction Pointers),</td>
</tr>
<tr>
<td></td>
<td>renaming logic, some smaller resources (ITLB, Streaming</td>
</tr>
<tr>
<td></td>
<td>Buffers, Return Stack Buffer, Branch History Buffer)</td>
</tr>
<tr>
<td><strong>Partitioned</strong></td>
<td>Uop Queue, Memory Instruction Queue, Re-Order Buffer,</td>
</tr>
<tr>
<td></td>
<td>General Instruction Queue, Load/Store buffers</td>
</tr>
</tbody>
</table>

Cache is shared & some other resources are partitioned? our approach is to run two cooperative threads (Main+Helper) of same application

Source: Intel Technology Journal’02
Software Architecture of Intel Compiler

C++ Front End  FORTRAN90 Front End

Profiler

Interprocedural Analysis & Optimizations

AutoHelper

Global Scalar Optimizations

IA-32 Back End  Itanium Back End
AutoHelper Tool

AutoHelper: End-to-end fully automated

VTune™ analyzer-based Delinquent Load Identification

Analysis: Loop Selection, Slicing, Triggering

Code Generation for Helper Threads
Pass 1: VTune analyzer-based Delinquent Load Identification

1. Run Intel VTune analyzer on a binary to collect cache-miss & clock-tick profiles.
   - Just need standard line# info in the binary.
   - Application can be an optimized binary
   - No special instrumentation pass needed.

2. Compiler reads in VTune analyzer *tb5* samples & correlate them back to Intel Compiler’s IR
   - Correlate using line#

3. Top loads with many clock ticks = Delinquent
Pass 2: Analysis for A Given Load

1. Select a loop for precomputation
   - On real machines, cost of thread activation/deactivation > 1k cycles
     - Should go for outer loop
   - On HT, some resources are shared/partitioned
     - Find loop with \textit{min resource requirement} & \textit{min #live-in}
     - Should go for inner loop with few live-ins & deactivate helper thread at end of loop to relinquish resource
   - Our algorithm: bottom-up traversal of loop graph
     - Greedy algorithm: Traversal ends when current loop is reasonably large & its outer loop doesn’t improve on the issues above

2. Use Basic SP for slicing within selected loop
   - Slicing is precise enough: Use memory disambiguation module in Intel compiler [Lavery & Ghiya. PLDI’01]
Pass 3: Codegen for Helper Threads

- Since a processor with HT Technology has 2 logical processors:
  - Create 1 helper thread in the beginning of execution
  - Activate/deactivate helper when entering/exiting a target loop

- Build Thread Graph to map slice to *multiple-entry threading* [Tian et al. ITJ’02]
- No conventional outlining
- Live-ins: Generate code for capture-private

\[ \text{ST} \xrightarrow{\text{activate}} \text{Hyper-Threading} \xrightarrow{\text{deactivate}} \text{ST} \xrightarrow{\text{activate}} \text{HT} \xrightarrow{\text{deactivate}} \ldots \]
# Experimental Environment

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.66 GHz Intel Pentium 4 Processor</td>
</tr>
<tr>
<td>L1 Trace Cache</td>
<td>12K micro-ops, 8-way set associative, 6 micro-ops per line</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16KB, 8-way set associative, 64-byte line, 2-cycle Int access, 4-cycle FP access, write through</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>512KB, 8-way set associative, 64-byte line, 7-cycle access</td>
</tr>
<tr>
<td>Load buffers</td>
<td>48 entries</td>
</tr>
<tr>
<td>Store buffers</td>
<td>24 entries</td>
</tr>
<tr>
<td>ROB</td>
<td>128 entries</td>
</tr>
<tr>
<td>OS</td>
<td>Windows® XP Professional, Service Pack 1</td>
</tr>
</tbody>
</table>
Case Study: MST
Case Study: MST Application in Olden Benchmark Suite

- As shown, helper thread executes 10% of instructions but covers ~60% cache misses.
  - 7.9% speedup

- Thread activation/deactivation mechanism: prototype hardware-based
  - Key to have this light-weight mechanism
  - If using heavier-weight Windows API (SetEvent & WaitForSingleObject), only 5.7% speedup

Source: Microprocessor Research Labs
Case Study: MCF Application in SPEC CINT2000 Suite

- Helper thread covers ~50% of cache misses
- 8.5% speedup

Thread activation/deactivation mechanism: prototype hardware-based

Synchronize with main thread every fixed number of iterations: Prototype hardware-based mechanism

- Key to have this light-weight mechanism
- If use heavier-weight synchronization, only 2.7% speedup

Source: Microprocessor Research Labs
Summary of Part II

❖ Can systematically generate helper threads to cover cache misses & get speedups on HT

❖ Possible future directions:

❖ Should have lighter-weight thread activation/deactivation? Better speedups

❖ Deactivation relinquishes resources to main thread on Hyper-Threading processors

❖ Can deactivate helpers aggressively & dynamically, because helper threads do not modify architectural states

❖ Improve compiler to construct optimized helpers that consume less computation resource on HT

❖ Trade-off computation & communication, loop unrolling etc.