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.

[redacted]

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**

[redacted]

Scope: Data Prefetching Threads

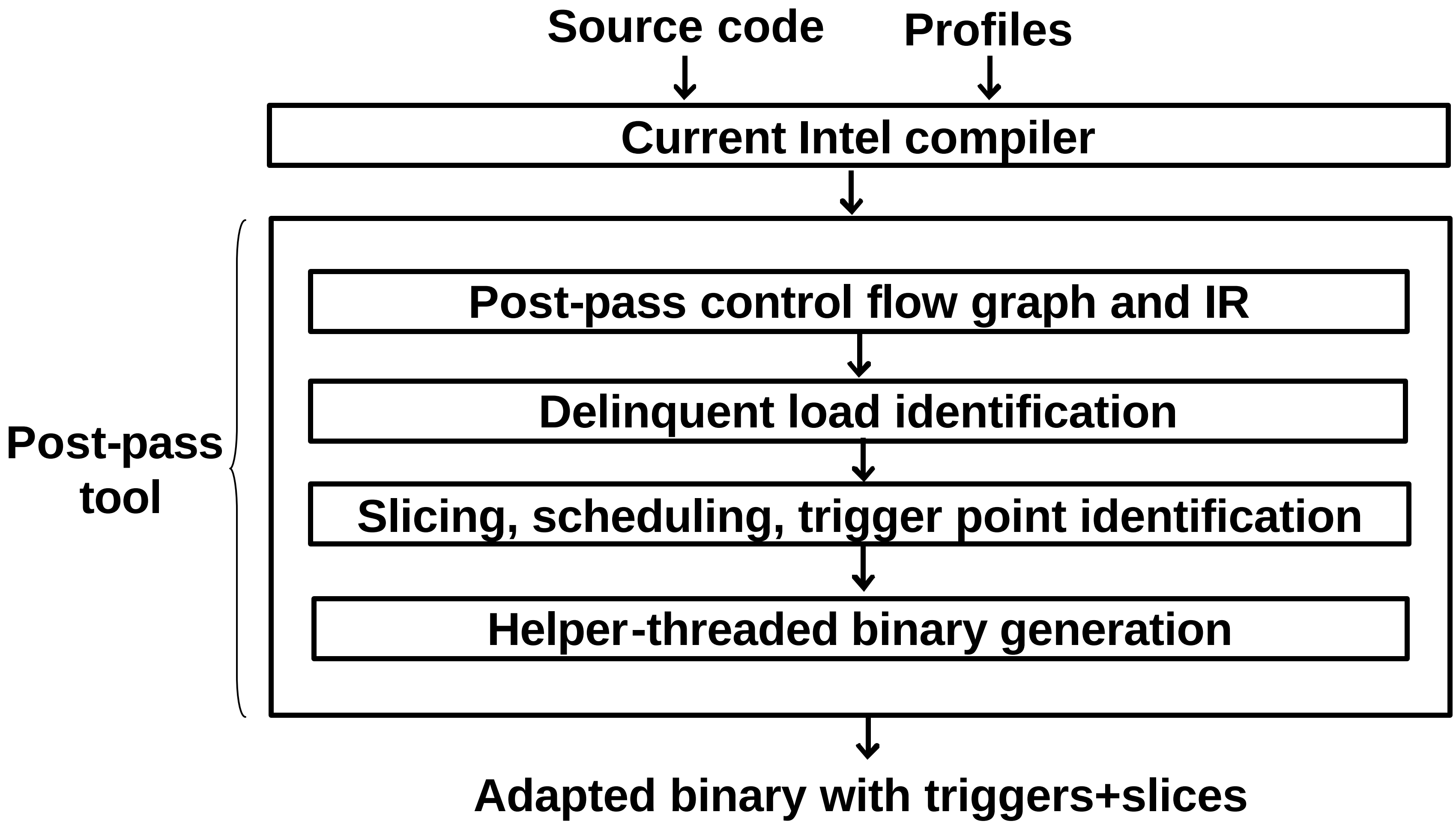
- **Improve data latency of single-threaded codes using multithreading:**
  - **Use additional thread to prefetch for the main thread**
  - **Use program itself as predictor, instead of address pattern predictor**

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**

[redacted]

Binary-Level Tool



Source code

Profiles

Current Intel compiler

Post-pass tool

Post-pass control flow graph and IR

Delinquent load identification

Slicing, scheduling, trigger point identification

Helper-threaded binary generation

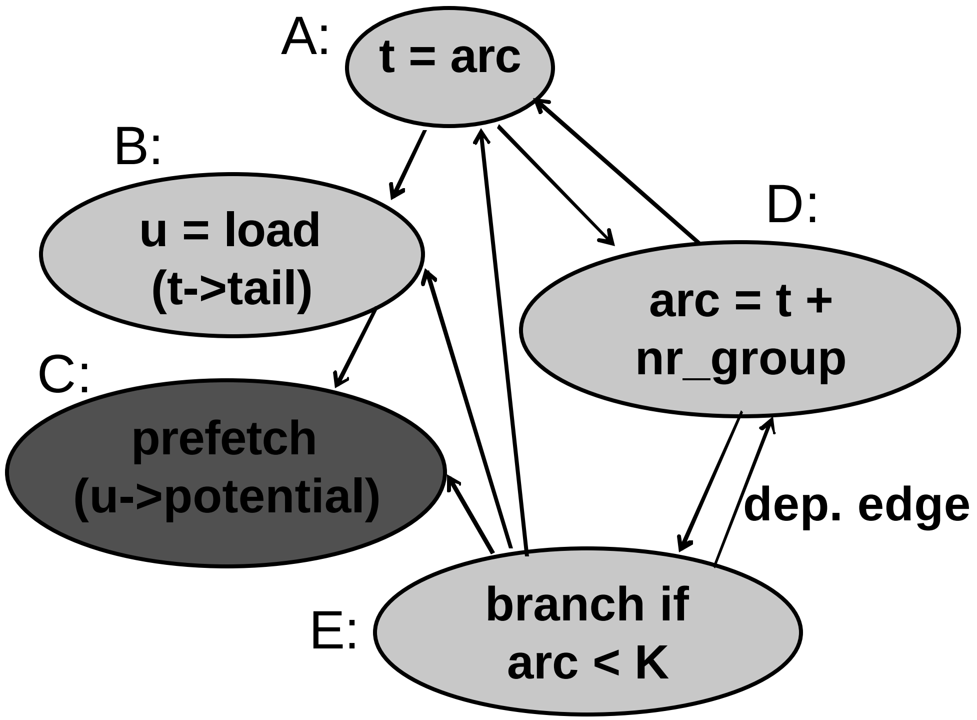
Adapted binary with triggers+slices

Basic SP: 1 helper thread does it all

slice

scheduling

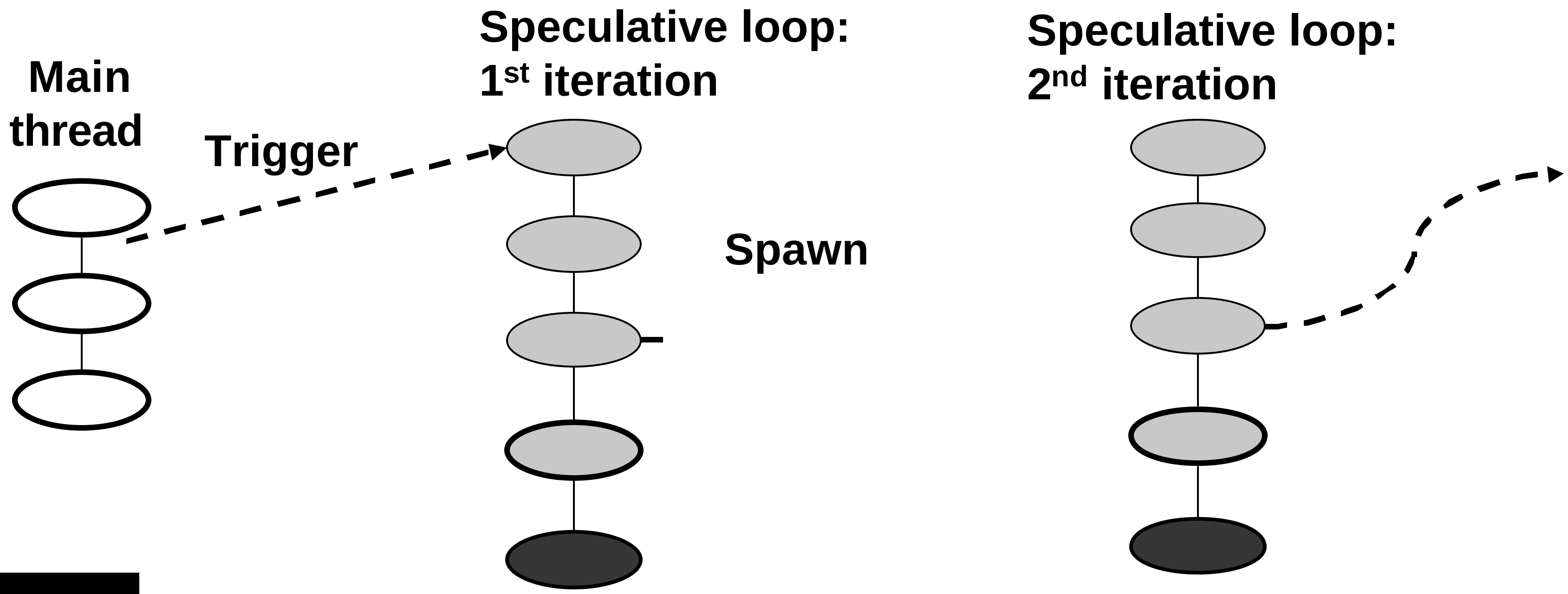
p-slice code for 1 helper thread



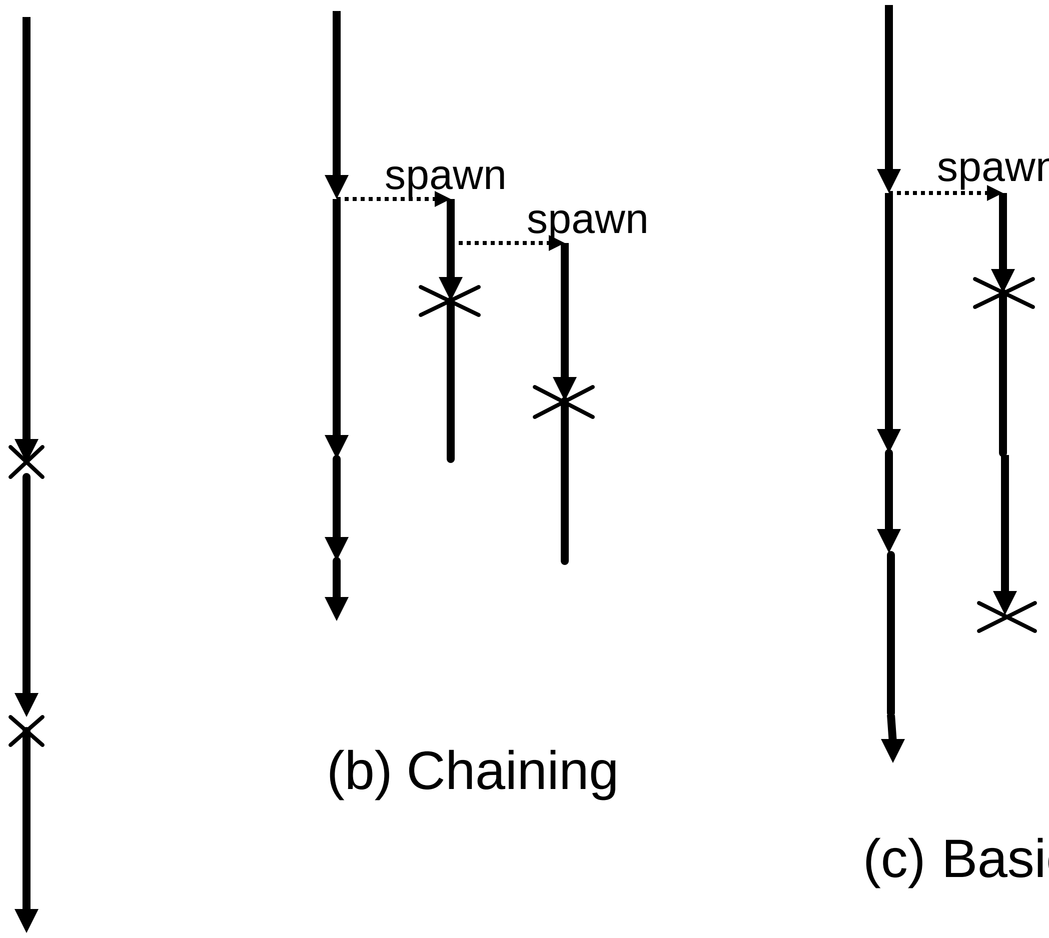
```
     do {
A:     t = arc;
B:     u = load(t->tail);
C:     prefetch(u->potential);
D:     arc = t + nr_group;
E:   } while (arc < K);
```

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
Trigger
Speculative loop: 1st iteration
Spawn
Speculative loop: 2nd iteration

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



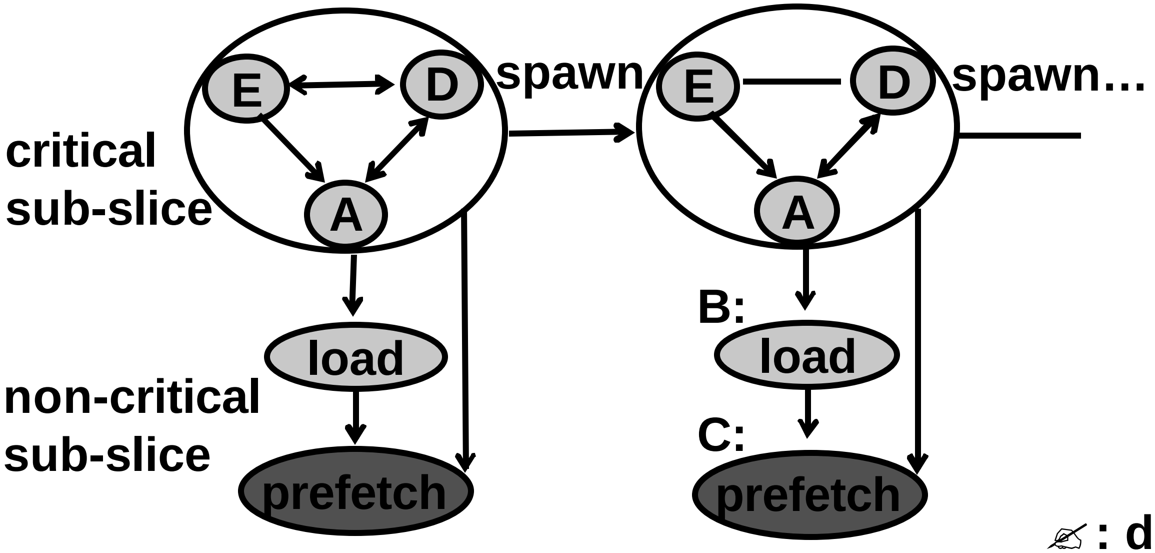
(a) Original

(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



✍ : dependence edge

Critical Slice in Doacross Loop

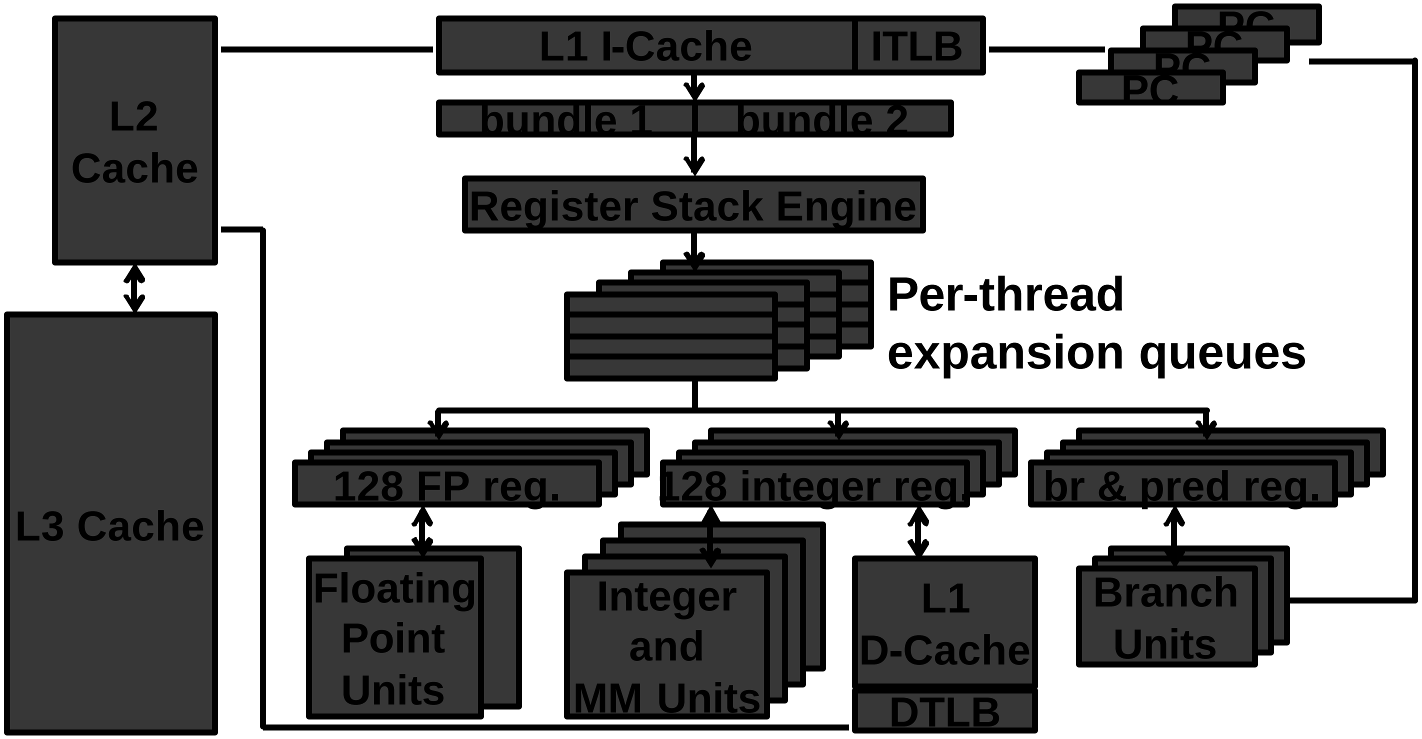
```
L1:
A:  t = arc;
D:  arc = t + nr_group;
E:  if (arc < K) spawn(L1);
B:  u = load(t->tail);
C:  prefetch(u->potential);
```

critical sub-slice (A, D, E)

non-critical sub-slice (B, C)

- **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 mechinism at user-level. (*chk*)**
  - **Live-in copy: use on-chip memory buffer for Register Stack Engine.**

Modeled Itanium Details

threading	SMT processor with 4 threads
pipelining	In-order: 12-stage. OOO: 16-stage
fetch,issue/cycle	2 bundles from 1 thread or 1 bundle each from 2 threads
window	In-order: 16-bundle expansion queue/thread. OOO: 255-entry reorder buffer/thread. 18-entry reservation station
registers/thread	128 integer, 128 FP, 64 predicates, 128 control, 8 branch
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

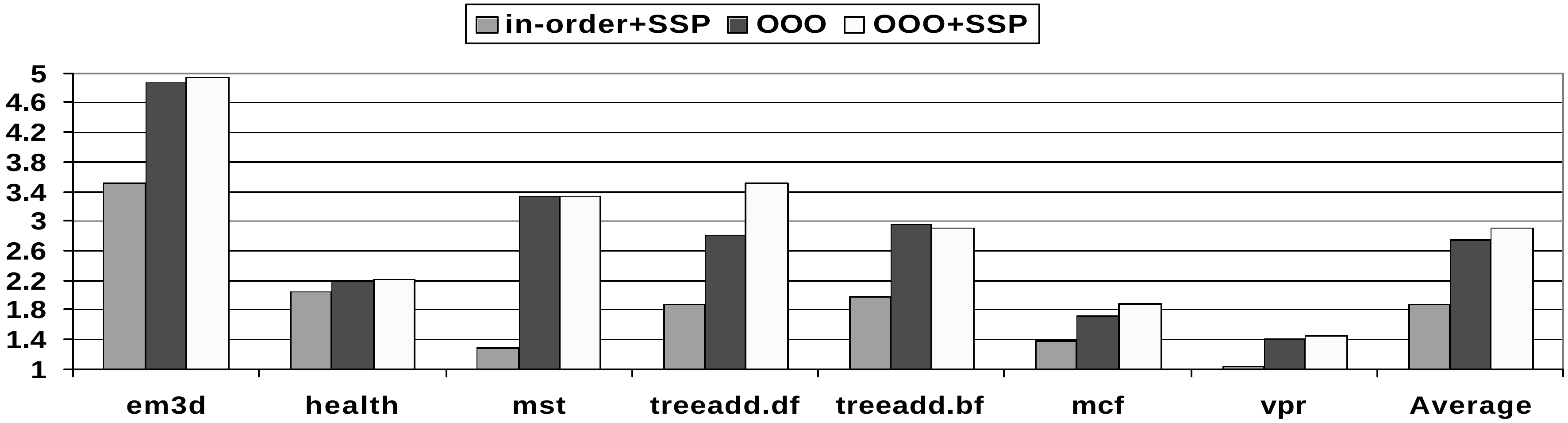
benchmark	slices (#)	average size	average # live-in	rely on
em3d	8	10.3	2.8	chaining
health	2	9.0	3.5	chaining
mst	4	28.3	4.8	chaining
treeadd.df	3	11.3	3.0	basic
treeadd.bf	2	12.5	4.5	chaining
mcf	5	14.0	4.4	chaining
vpr	6	13.5	4.0	chaining

- **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

[redacted]

Speedup on in-order & OOO models

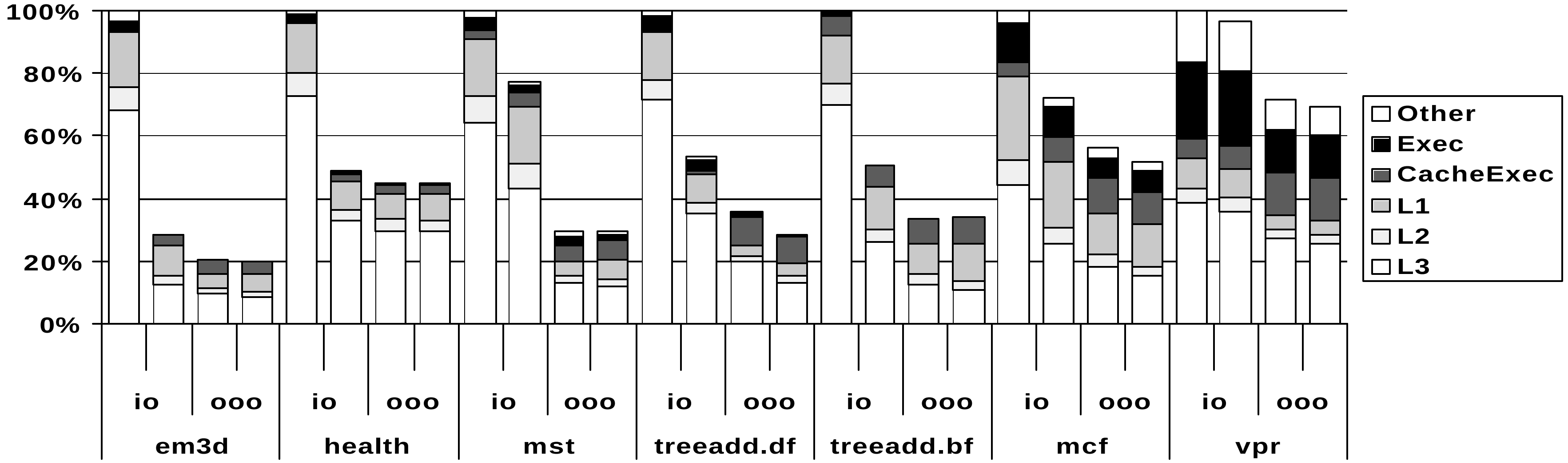


- **SSP: Our Software-based SP**
- **Baseline: In-order processor without SSP**
- **On in-order: SSP improves 87%.**
- **On OOO: SSP improves 5%**

Source: Liao, PLDI'02

[redacted]

Cache Latency Reduction Analysis



- ✍ **Long-range prefetching ✍ SSP 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

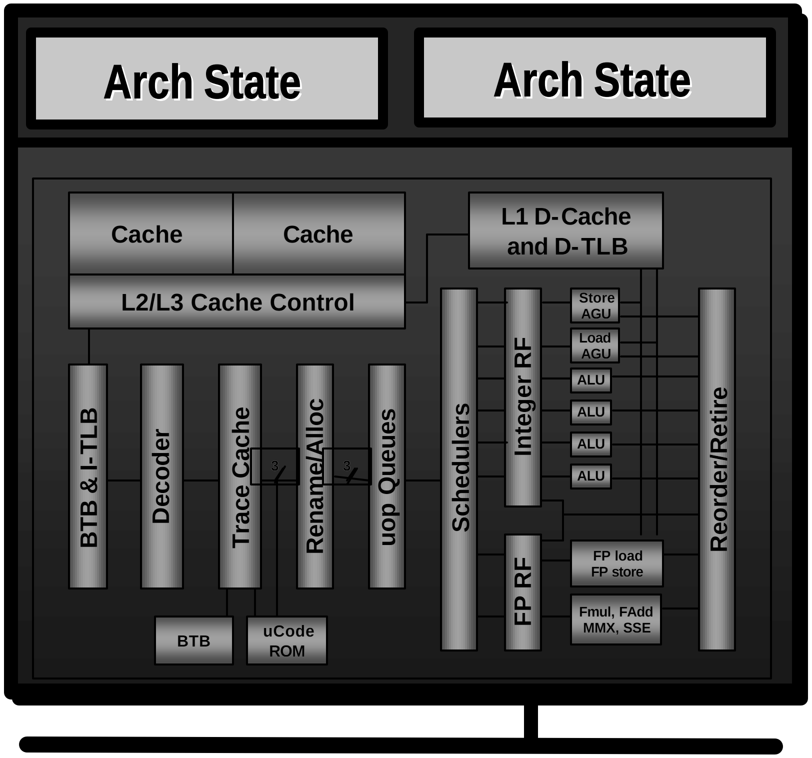
- **Motivation for AutoHelper study on IA-32**
  - **Arrival of Pentium 4 Processors with HT Technology**
    ? **Evaluate simulator-based ideas**
  - **If manually constructing helper thread's code:**
    - **Error-prone**
    - **Not providing systematic study or insight on HT Technology**

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

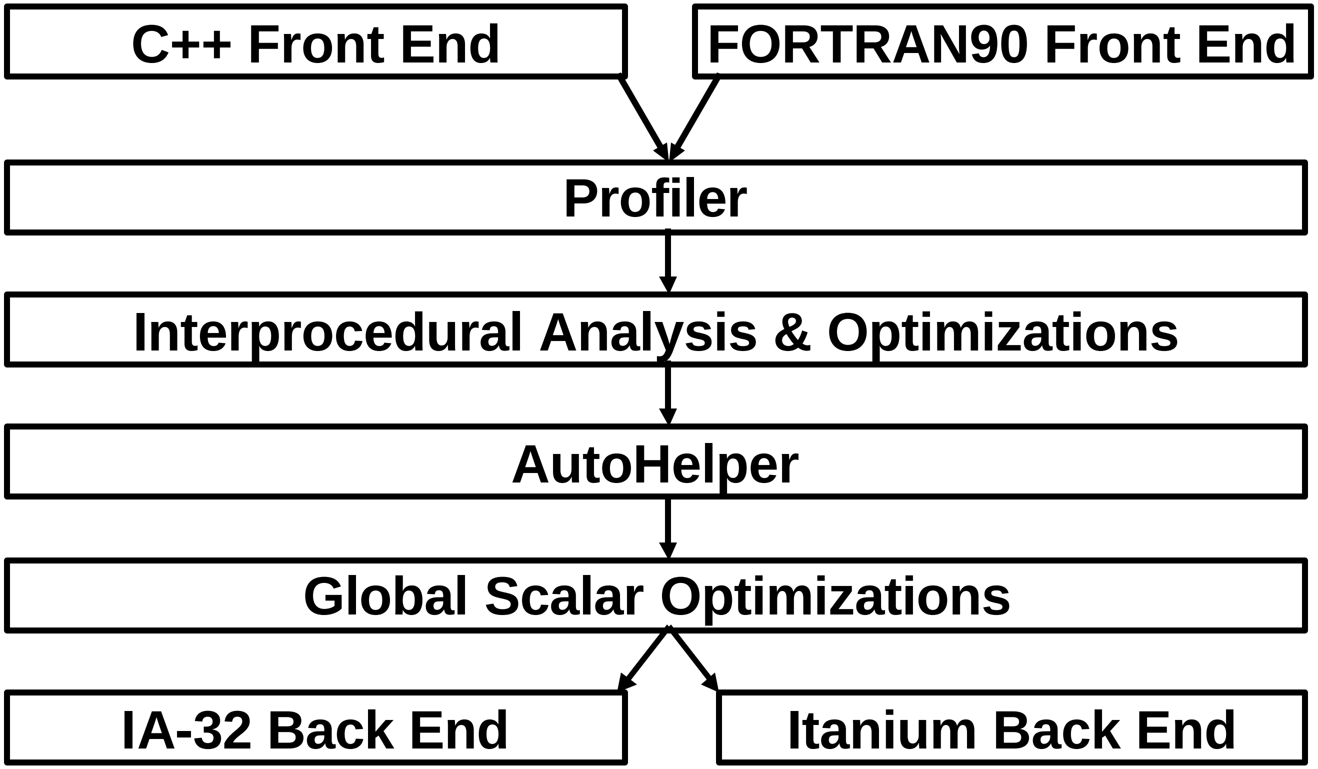
Hardware Management in Processors with HT Technology

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

✍ **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

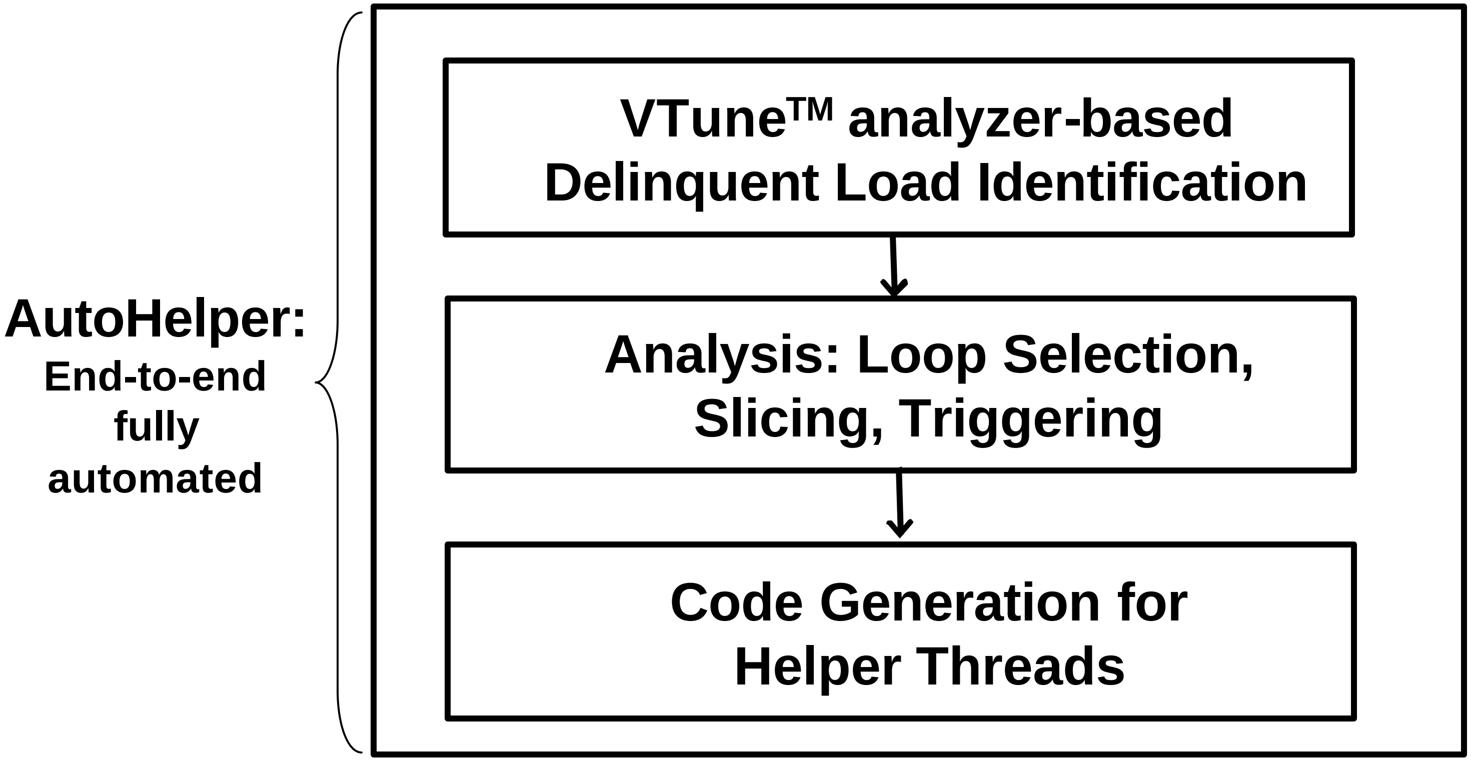
AutoHelper

Global Scalar Optimizations

IA-32 Back End

Itanium Back End

AutoHelper Tool



AutoHelper:
End-to-end fully automated

VTune^{TM} 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 *min resource requirement & 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**

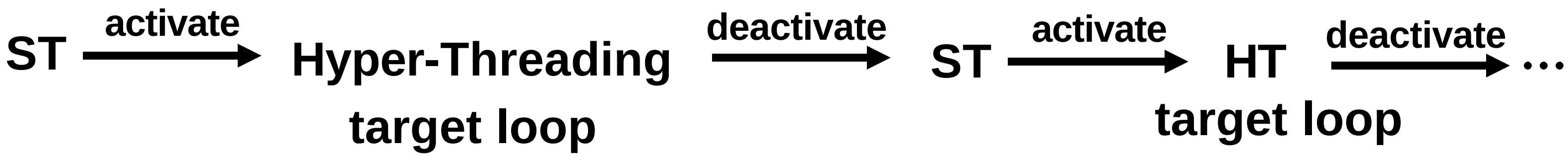
     - ✍ **Greedy algorithm: Traversal ends when current loop is reasonably large & its outer loop doesn't improve on the issues above**

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**



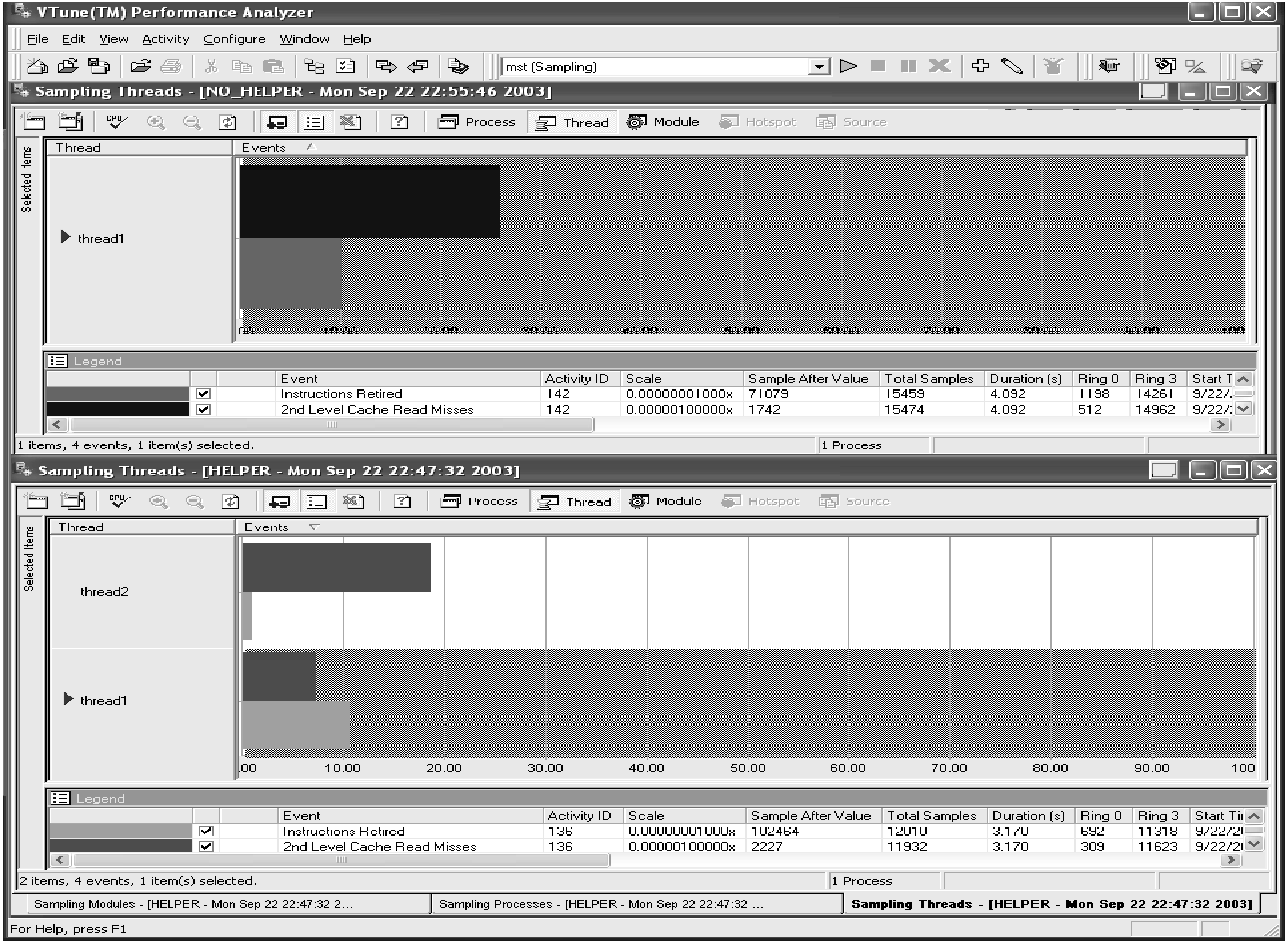
ST —activate→ **Hyper-Threading target loop** —deactivate→ **ST** —activate→ **HT target loop** —deactivate→ **…**

- **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**

Experimental Environment

CPU	**2.66 GHz Intel Pentium 4 Processor**
L1 Trace Cache	**12K micro-ops, 8-way set associative 6 micro-ops per line**
L1 Data Cache	**16KB, 4-way set associative, 64-byte line, 2-cycle Int access, 4-cycle FP access, write through**
L2 Unified Cache	**512KB, 8-way set associative, 64-byte line, 7-cycle access**
DTLB	**64 entries, fully associative, map a 4KB-page**
Load buffers	**48 entries**
Store buffers	**24 entries**
ROB	**128 entries**
OS	**Windows® XP Professional, Service Pack 1**

Case Study: MST



Sampling Threads - [NO_HELPER - Mon Sep 22 22:55:46 2003]

			Event	Activity ID	Scale	Sample After Value	Total Samples	Duration (s)	Ring 0	Ring 3	Start T
▨	☑		Instructions Retired	142	0.00000001000x	71079	15459	4.092	1198	14261	9/22/:
■	☑		2nd Level Cache Read Misses	142	0.00000100000x	1742	15474	4.092	512	14962	9/22/:

1 items, 4 events, 1 item(s) selected. 1 Process

Sampling Threads - [HELPER - Mon Sep 22 22:47:32 2003]

			Event	Activity ID	Scale	Sample After Value	Total Samples	Duration (s)	Ring 0	Ring 3	Start Ti
▨	☑		Instructions Retired	136	0.00000001000x	102464	12010	3.170	692	11318	9/22/2
■	☑		2nd Level Cache Read Misses	136	0.00000100000x	2227	11932	3.170	309	11623	9/22/2

2 items, 4 events, 1 item(s) selected. 1 Process

Sampling Modules - [HELPER - Mon Sep 22 22:47:32 2... | Sampling Processes - [HELPER - Mon Sep 22 22:47:32 ... | Sampling Threads - [HELPER - Mon Sep 22 22:47:32 2003]

For Help, press F1

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.**

[redacted]