The Bulk Multicore Architecture for Programmability

Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Acknowledgments

Key contributors:

- Luis Ceze
- Calin Cascaval
- James Tuck
- Pablo Montesinos
- Wonsun Ahn
- Milos Prvulovic
- Pin Zhou
- YY Zhou
- Jose Martinez

Arch Challenge: Enable a Programmable Environment

- Able to attain high efficiency while relieving the programmer from low-level tasks
- Help minimize chance of (parallel) programming errors

The Bulk Multicore [CommACM 09]

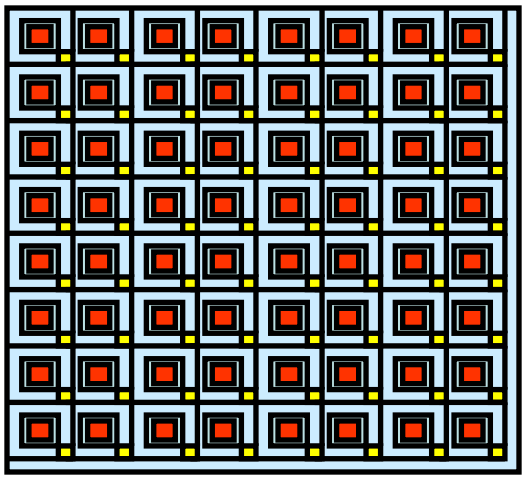
http://iacoma.cs.uiuc.edu/bulkmulticore.pdf

General-purpose multicore for programmability

- Novel scalable cache-coherent shared-memory (signatures & chunks)
  - Relieves programmer/runtime from managing shared data
- High-performance sequential memory consistency
  - Provides a more SW-friendly environment
- HW primitives for a low-overhead program dev & debug environment
  (data-race detection, deterministic replay, address disambiguation)
  - Helps reduce the chance of parallel programming errors
  - Overhead low enough to be "on" during production runs

The Bulk Multicore

- Idea: Eliminate the commit of individual instructions at a time
- Mechanism:
  - By default, processors commit chunks of instructions at a time (e.g. 2,000 dynamic instr)
  - Chunks execute atomically and in isolation (using buffering and undo)
  - Memory effects of chunks summarized in HW signatures
  - Chunks invisible to SW
- Advantages over current:
  - Higher programmability
  - Higher performance
  - Simpler processor hardware



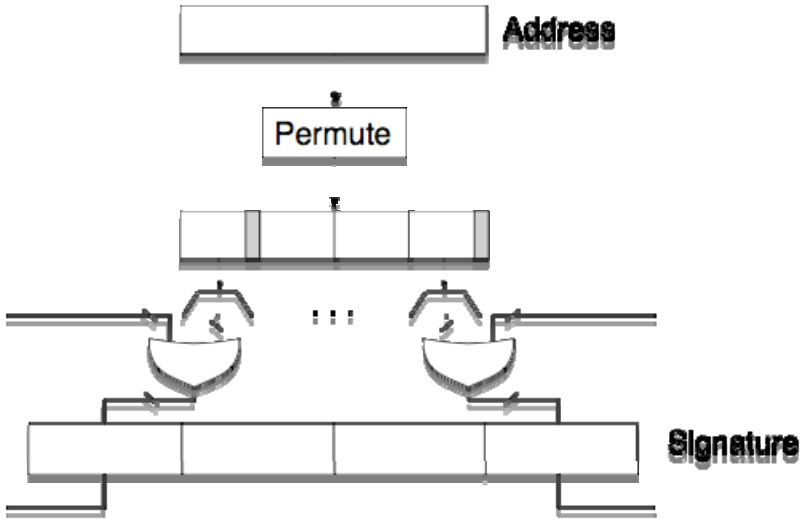
The Bulk Multicore

Rest of the Talk

- The Bulk Multicore
- How it improves programmability

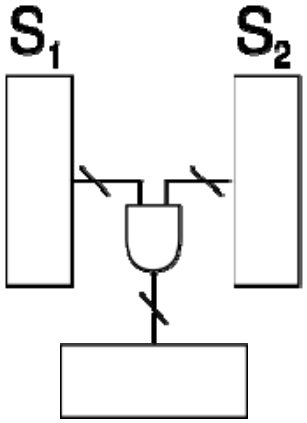
Hardware Mechanism: Signatures [ISCA06]

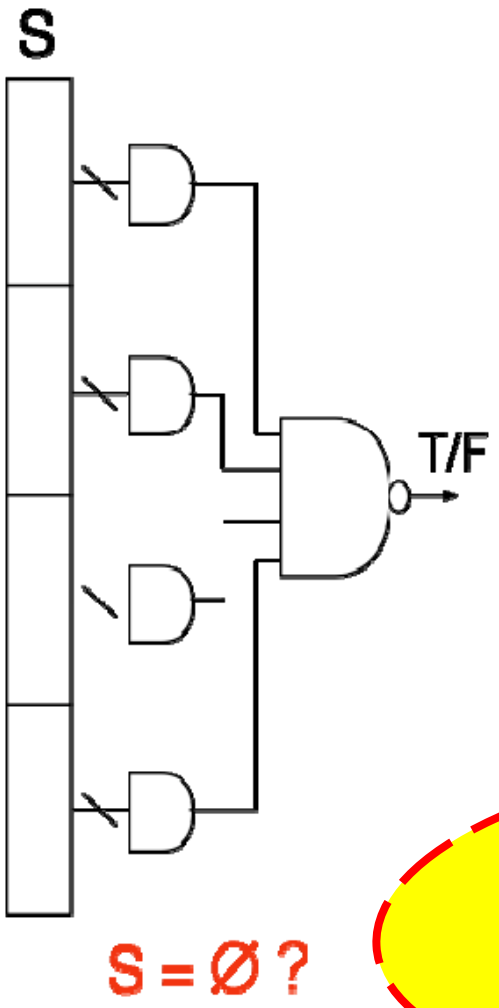
- Hardware accumulates the addresses read/written in signatures

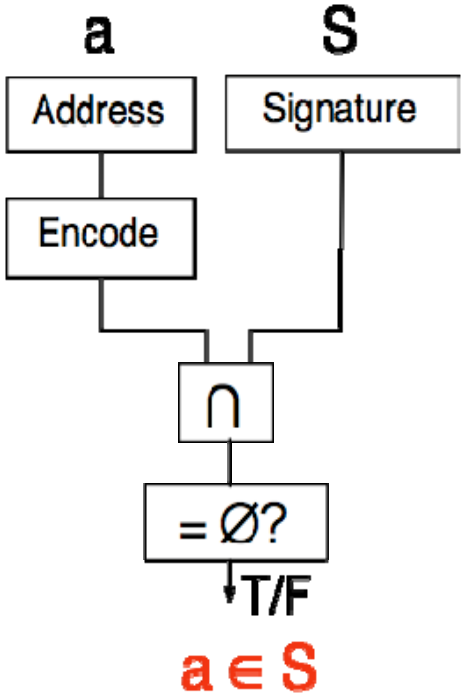


  - Read and Write signatures
  - Summarize the footprint of a Chunk of code

Signature Operations In Hardware

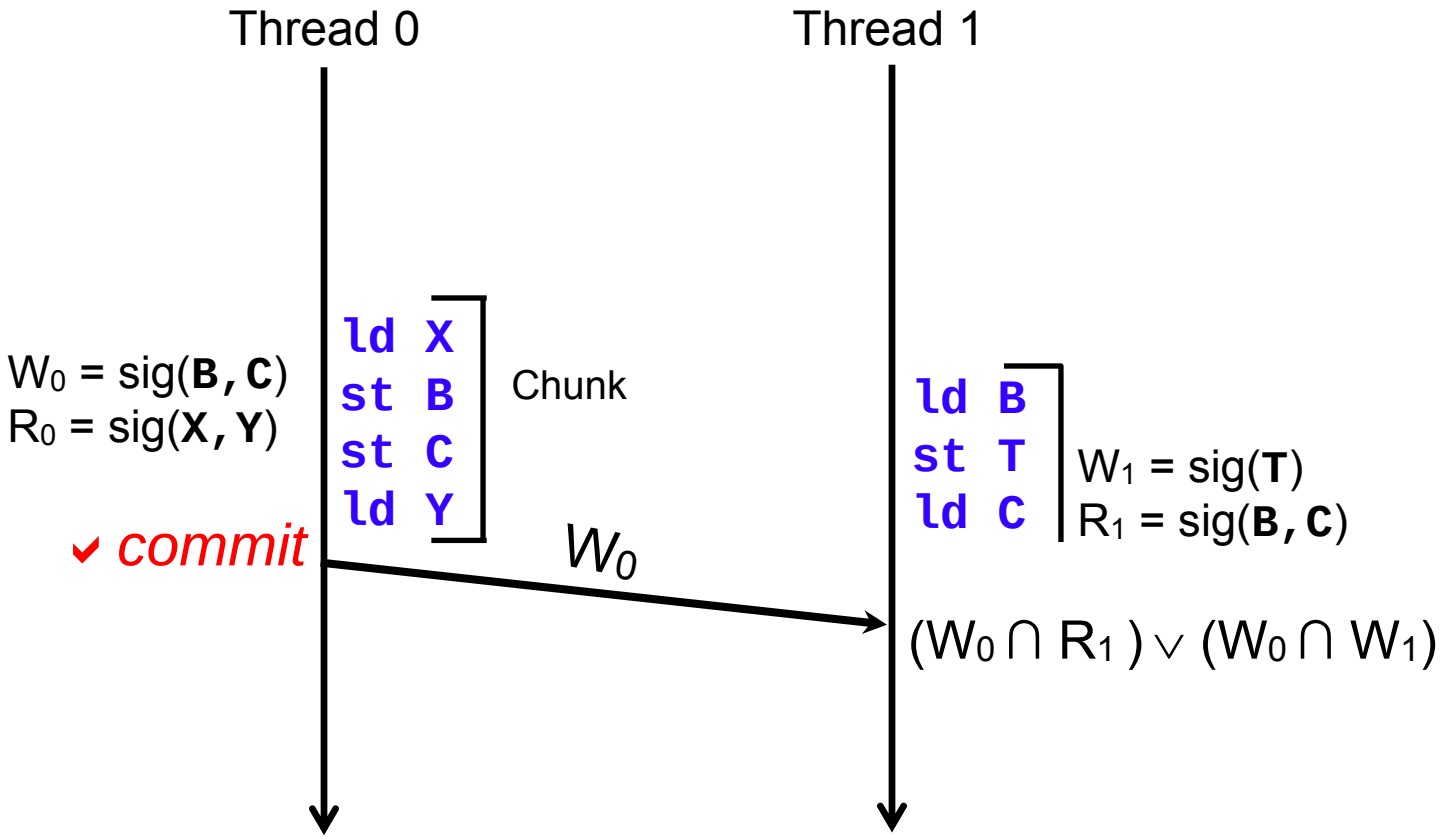






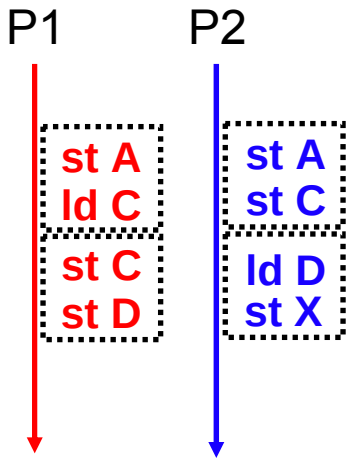
Inexpensive Operations on Groups of Addresses

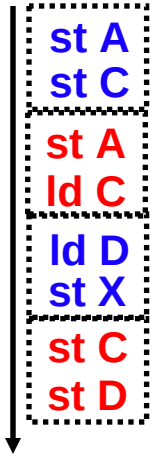
Executing Chunks Atomically & In Isolation: Simple!

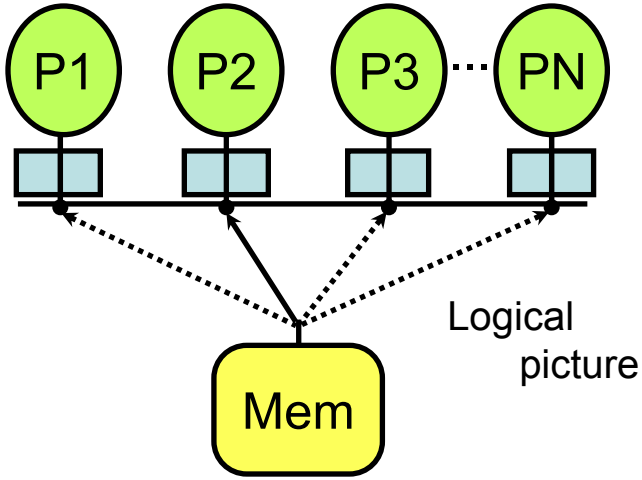


Chunk Operation + Signatures: Bulk [ISCA07]

- Execute each chunk atomically and in isolation
- (Distributed) arbiter ensures a total order of chunk commits







- Supports Sequential Consistency [Lamport79]:
  - Low hardware complexity: Need not snoop ld buffer for consistency
  - High performance: Instructions are fully reordered by HW
    loads and stores make it in any order to the sig
    Fences are NOOPS

Summary: Benefits of Bulk Multicore

- Gains in HW simplicity, performance, and programmability

- Hardware simplicity:
  - Memory consistency support moved away from core
  - Toward commodity cores
  - Easy to plug-in accelerators

- High performance:
  - HW reorders accesses heavily (intra- and inter-chunk)

Benefits of Bulk Multicore (II)

- High programmability:
  - Invisible to the programming model/language
  - Supports Sequential Consistency (SC)
    - * Software correctness tools assume SC
  - Enables novel always-on debugging techniques
    - * Only keep per-chunk state, not per-load/store state
      - * Deterministic replay of parallel programs with no log
      - * Data race detection at production-run speed

Benefits of Bulk Multicore (III)

- Extension: Signatures visible to SW through ISA
  - Enables pervasive monitoring
  - Enables novel compiler opts

Many novel programming/compiler/tool opportunities

Rest of the Talk

- The Bulk Multicore
- How it improves programmability

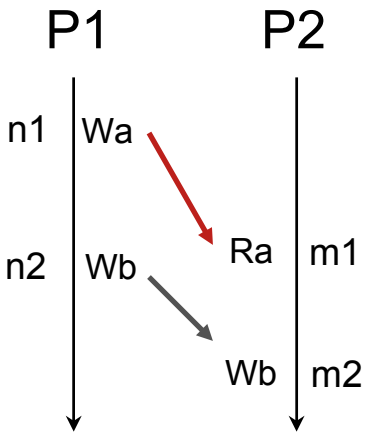
Supports Sequential Consistency (SC)

- Correctness tools assume SC:
  - Verification tools that prove software correctness
- Under SC, semantics for data races are clear:
  - Easy specifications for safe languages
- Much easier to debug parallel codes (and design debuggers)
- Works with “hand-crafted” synchronization

Deterministic Replay of MP Execution

- During Execution: HW records into a log the order of dependences between threads
- The log has captured the "interleaving" of threads
- During Replay: Re-run the program
  - Enforcing the dependence orders in the log

Conventional Schemes



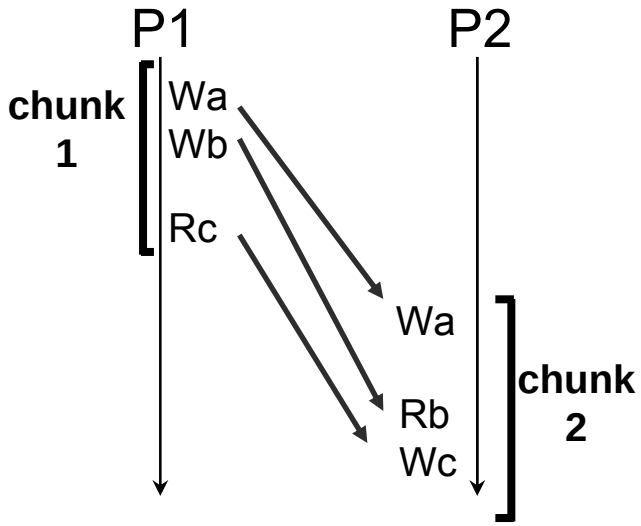
P2's Log

P1	n1	m1
P1	n2	m2

- Potentially large logs

Bulk: Log Necessary is Minuscule [ISCA08]

- During Execution:
  - Commit the instructions in chunks, not individually



Combined Log of all Procs:

P1

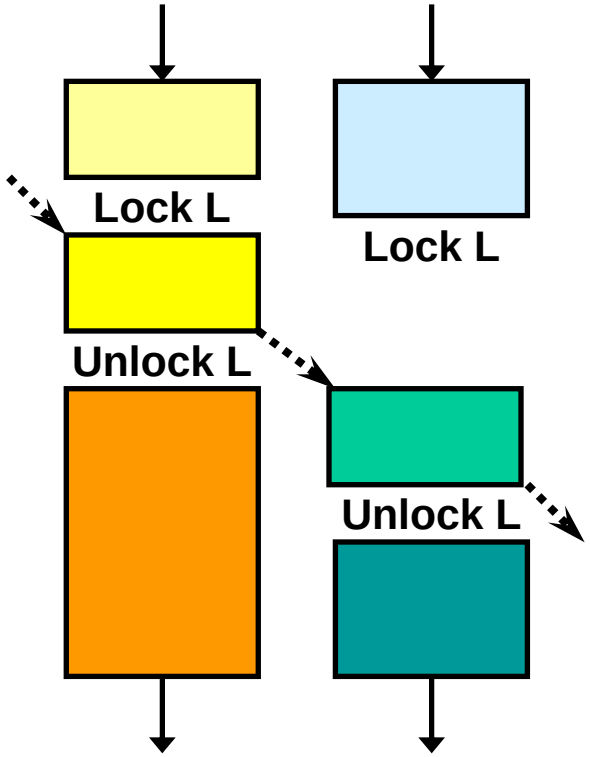
P2

Pi

If we **fix** the chunk commit interleaving:

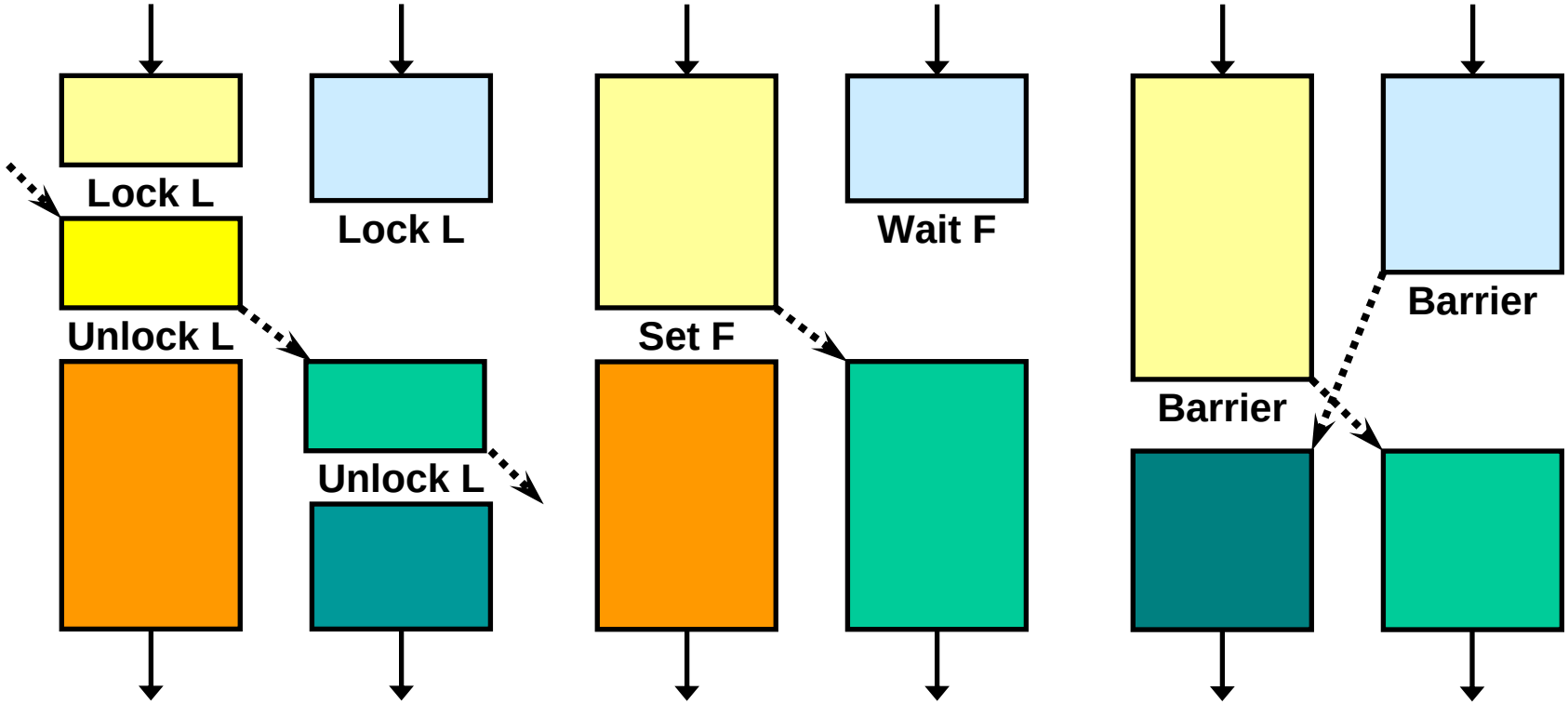
Combined Log = NIL

Data Race Detection at Production-Run Speed [ISCA03]



- If we detect communication between…
  - Ordered chunks: not a data race
  - Unordered chunks: data race

Different Synchronization Ops



Benefits of Bulk Multicore (III)

- Extension: Signatures visible to SW through ISA
  - Enables pervasive monitoring **[ISCA04]**

    Support numerous watchpoints for free
  - Enables novel compiler opts **[ASPLOS08]**

    Function memoization

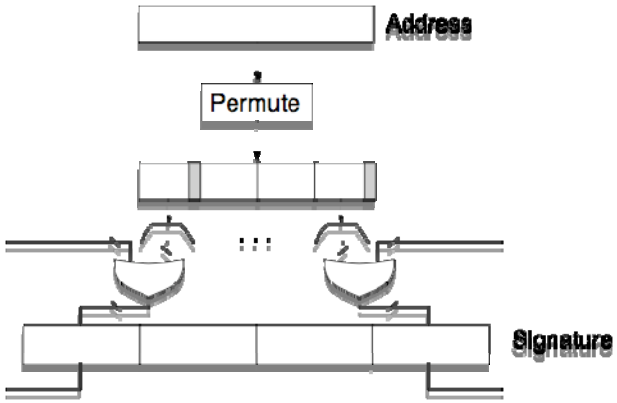
    Loop-invariant code motion

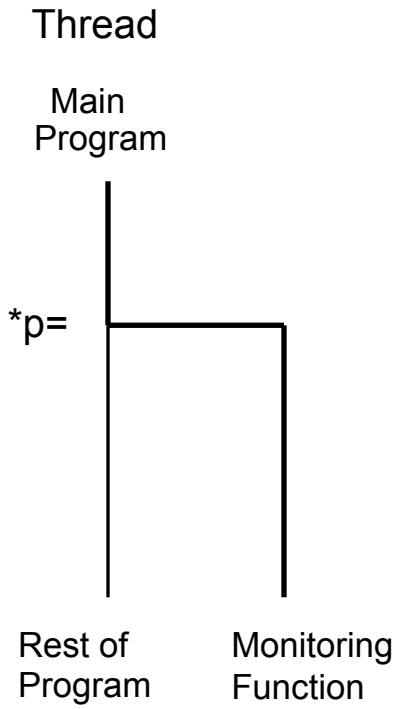
Pervasive Monitoring: Attaching a Monitor Function to Address

- Watch memory location
- Trigger monitoring function when it is accessed

```
        instr
Watch(addr, usr_monitor)
        instr
        instr
        instr
    *p = ...
        instr
        instr
        instr
```

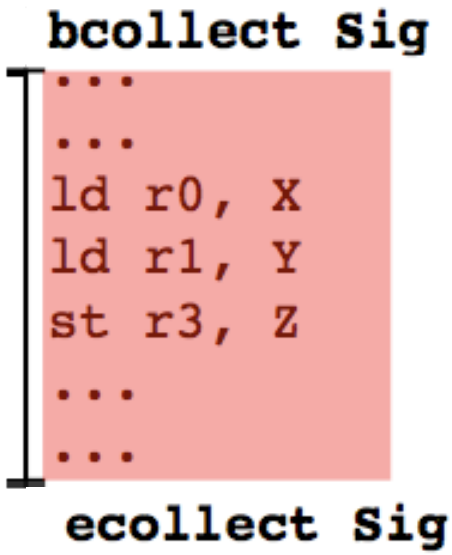
```
usr_monitor(Addr){
   …..
}
```





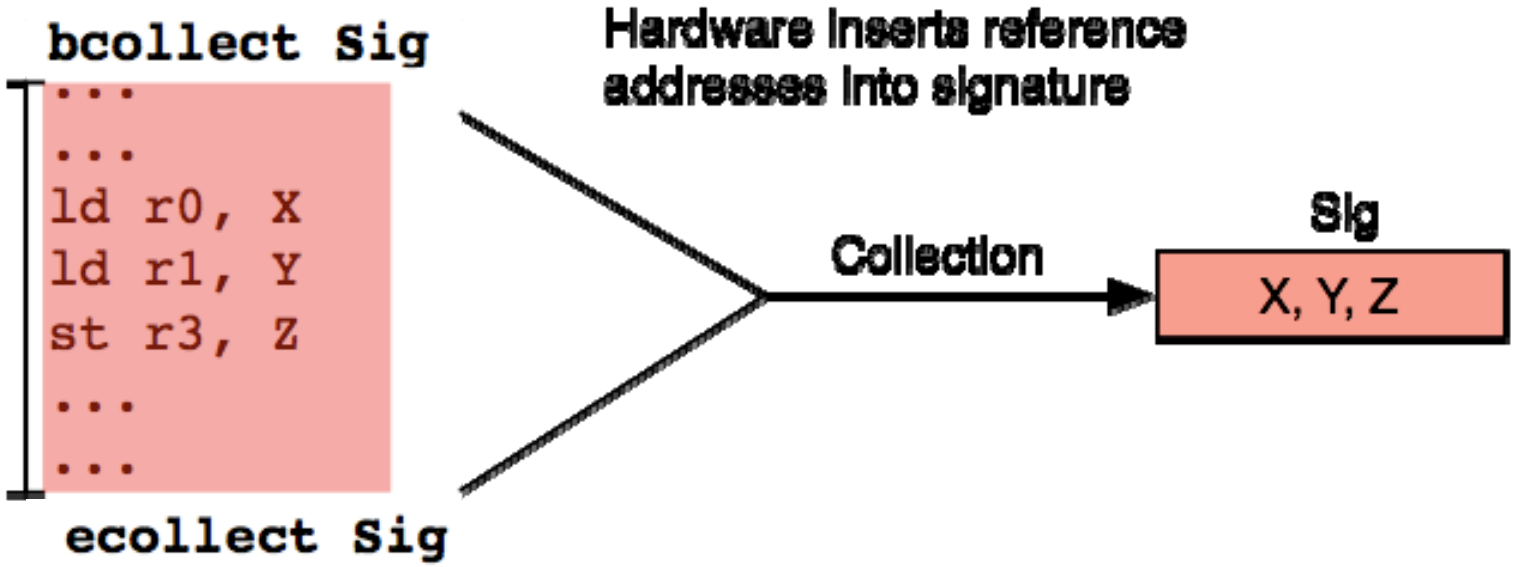
Enabling Novel Compiler Optimizations

New instruction: Begin/End collecting addresses into sig

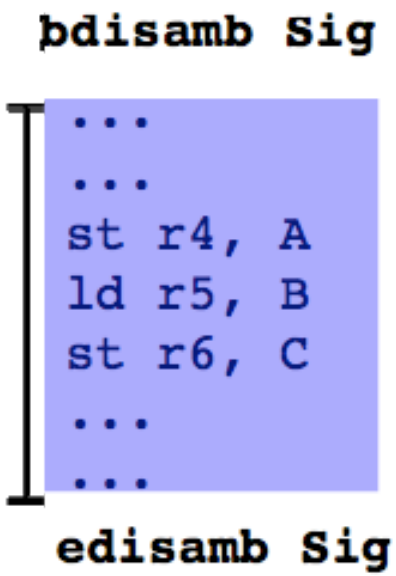


Enabling Novel Compiler Optimizations

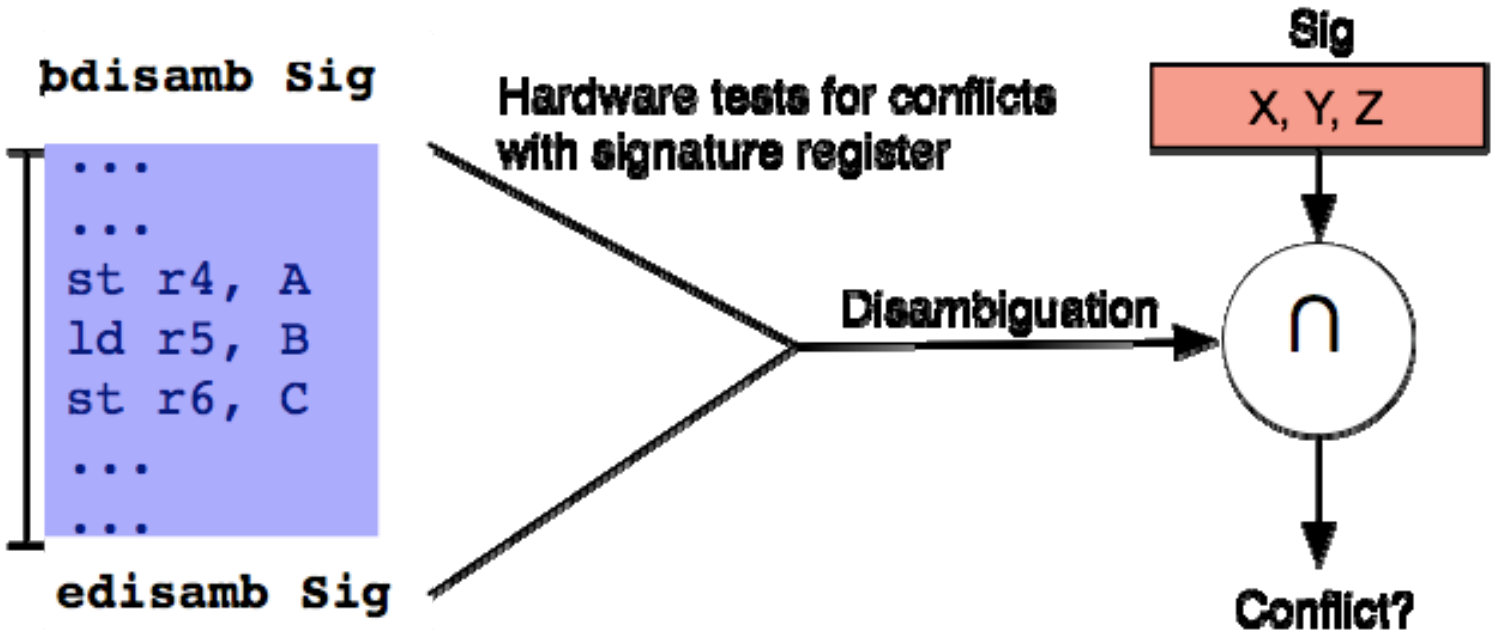
New instruction: Begin/End collecting addresses into sig



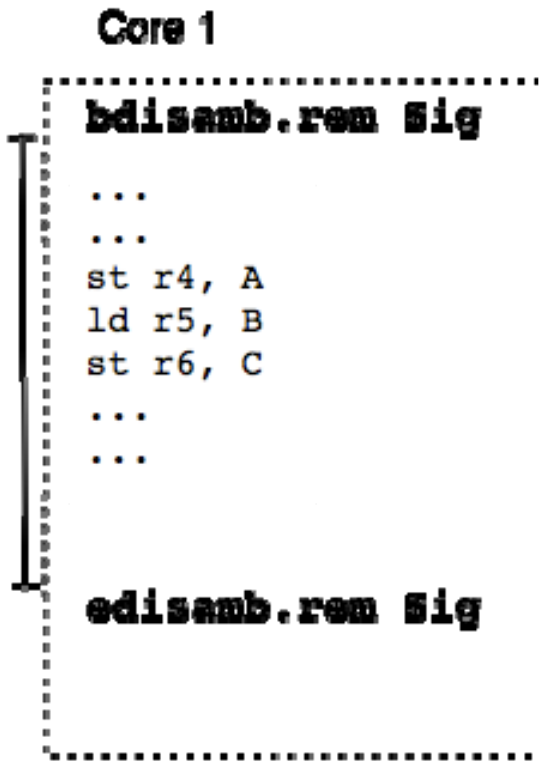
Instruction: Begin/End Disambiguation Against Sig



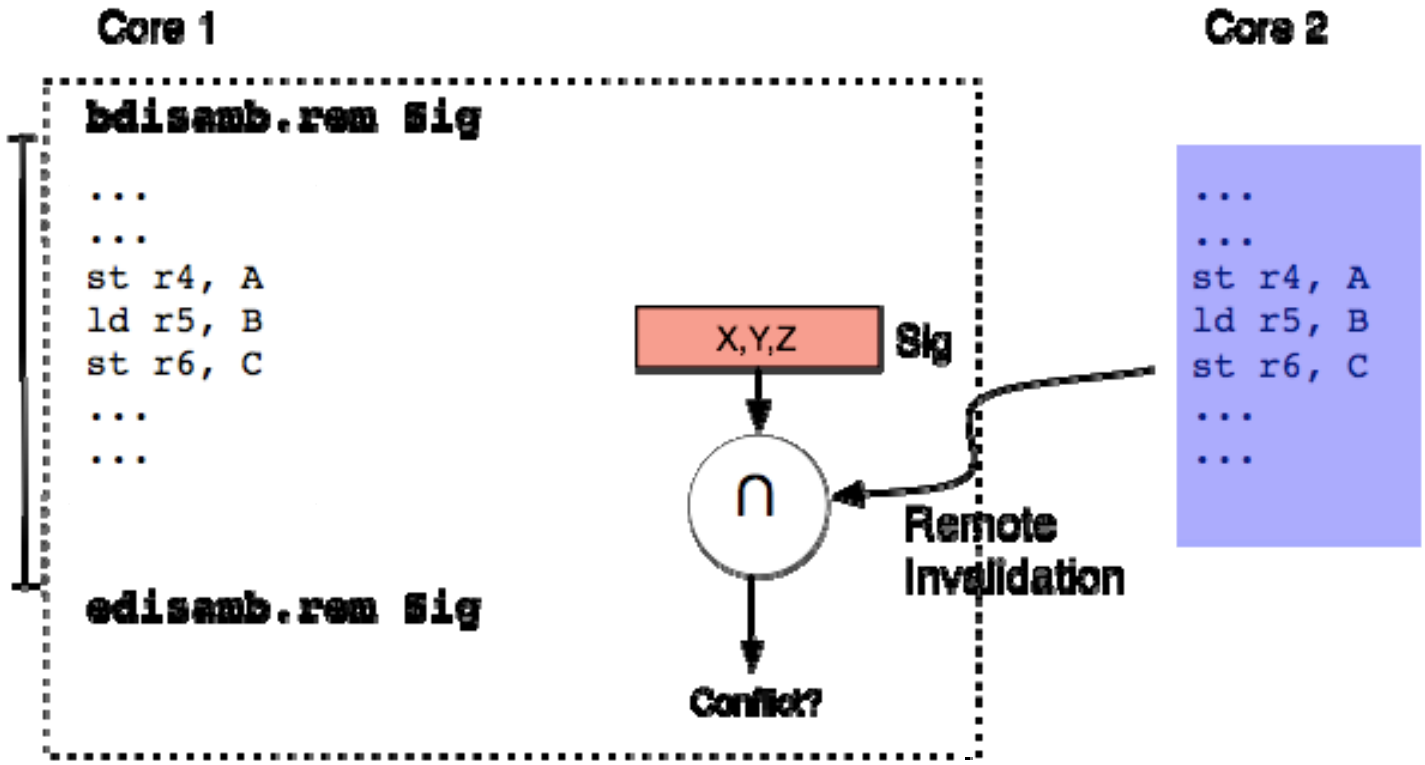
Instruction: Begin/End Disambiguation Against Sig



Instruction: Begin/End Remote Disambiguation



Instruction: Begin/End Remote Disambiguation



Optimization: Function Memoization

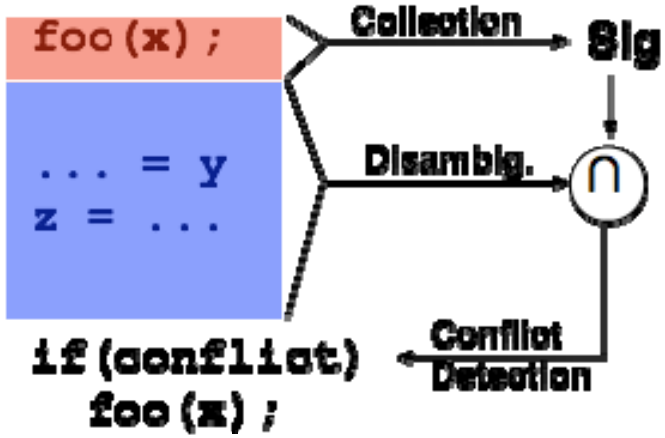
- Goal: skip the execution of functions

```
foo(x);

... = y
z = ...

foo(x);
```

Example Opt: Function Memoization

- Goal: skip the execution of functions whose outputs are known

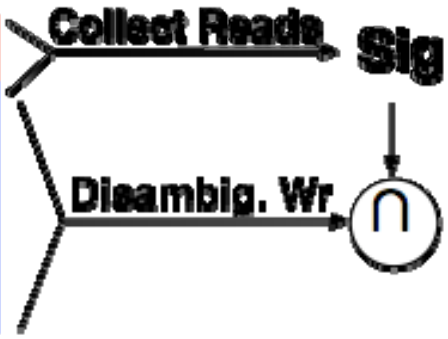
```
foo(x);

... = y
z = ...

foo(x);
```



Example Opt: Loop-Invariant Code Motion

```
while(...) {
    ...
    ... = <expr>
    ...
}
```

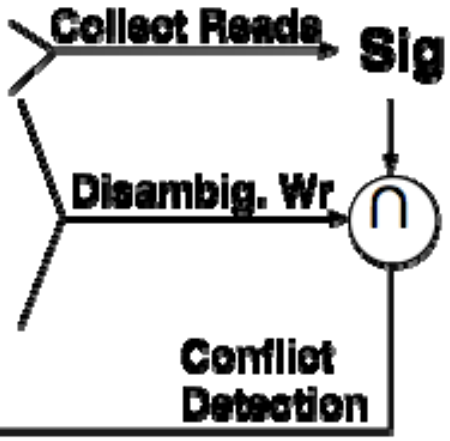
```
reg = <expr>
while(...) {
    ...
    ... = reg
    ...
}
```



Example Opt: Loop-Invariant Code Motion

```
while(...) {
    ...
    ... = <expr>
    ...
}
```

```
checkpoint()
reg = <expr>
while(...) {
    ...
    ... = reg
    ...
}
if (conflict)
  rollback()
  <original loop>
```



Summary:
The Bulk Multicore for Year 2015-2018

- 128+ cores/chip, coherent shared-memory (perhaps in groups)
- Simple HW with commodity cores
  - Memory consistency checks moved away from the core
- High performance shared-memory programming model
  - Execution in programmer-transparent chunks
  - Signatures for disambiguation, cache coherence, and compiler opts
- High programmability:
  - Sequential consistency
  - Sophisticated always-on development support
    - Deterministic replay of parallel programs with no log (*DeLorean*)
    - Data race detection for production runs (*ReEnact*)
    - Pervasive program monitoring (*iWatcher*)

The Bulk Multicore Architecture for Programmability

Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu