Explicitly Parallel Architectures for Memory Performance Enhancement

Christoforos E. Kozyrakis

Computer Science Division
University of California at Berkeley

kozyraki@cs.berkeley.edu
http://iram.cs.berkeley.edu
Thesis

- Memory performance enhancement requires the **synergy** of hardware and software techniques

- This talk: focus on hardware
  - **Explicitly parallel instruction sets** for efficient communication of parallelism between hardware and software
  - **Exploit memory bandwidth** to hide or tolerate high memory latency
Software-Hardware Strengths

- Software (compiler/run-time system)
  - Better view and understanding of both application requirements and system resources
  - Ability to apply complex optimizations over thousands of source-code lines

- Hardware
  - Massive resources for parallel execution of memory and arithmetic operations
  - Can use dynamic (run-time) information to apply aggressive optimizations within a small window of instructions
Problems with Current ISAs

- Strict sequential semantics
  - Sequential ordering even for independent ops
  - Any parallelism must be (re-)discovered by HW

- Lack of memory access pattern information
  - Any regular access pattern is broken into a series of accesses to a single memory location
  - Access pattern must be “(re-)discovered” by HW to apply optimizations (e.g. prefetching)

- Small number of registers
  - Exacerbate the memory performance problem
  - Make several compiler optimizations difficult
The Semantics Problem

- Starting with:
  
  ```c
  for (i=0; i<N; i++) *(A+10*i)++;
  ```

- You get (naive version):
  
  ```c
  Loop:    ld   t0, 0(A)
          addi  t0, t0, 1
          st    t0, 0(A)
          addi  A, A, 10
          addi  i, i, 1
          bne   i, N, Loop
  ```

  No stride info!
  No locality info!
  No mem dependence info
  Dependent operations!

- To get high performance you need
  - Unrolling, SW pipelining, prefetching, scheduling (SW)
  - Prefetching, stride predictor, dynamic scheduling, speculation, dynamic memory disambiguation (HW)
Hardware for Current ISAs

- ~80% of the die used to (re-)discover parallelism and address memory dependencies and latency
- Techniques like caching and speculation are becoming inefficient in resource utilization
- Poor match to media applications with real-time requirements and limited spatial locality of data
# Explicitly Parallel Instruction Sets

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify independent ops to be executed in parallel</td>
<td>Simpler instruction issue logic</td>
</tr>
<tr>
<td>Provide explicit memory access pattern info</td>
<td>Simper (pre)fetched decisions; effective cache management</td>
</tr>
<tr>
<td>Provide large number of registers</td>
<td>Less memory accesses; enable SW prefetching; less memory dependencies</td>
</tr>
<tr>
<td>Support for predication, load/store speculation</td>
<td>More aggressive scheduling of load/store ops; simpler HW for memory dependencies and exceptions (faults)</td>
</tr>
</tbody>
</table>
Parallel Semantics

- Starting with:
  \[
  \text{for (i=0; i<N; i++) *(A+10*i)++;}
  \]

- You want to get:
  
  ```
  Loop:  vld.nc v0, A, 10, L;  addi i, i, L
          vaddi v0, v0, 1
          vst.nc v0, A, 10, L;  addi A, A, 10L
          bne i, N, Loop
  ```

Vector load/store; stride 10; L elements; no-caching

Independent operations; execute in parallel
## Explicit ISAs & Systems

<table>
<thead>
<tr>
<th>Example</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA-64 (EPIC)</td>
<td>- Intel &amp; HP</td>
</tr>
<tr>
<td></td>
<td>- VLIW with 128 registers</td>
</tr>
<tr>
<td></td>
<td>- Memory accesses with locality hints</td>
</tr>
<tr>
<td></td>
<td>- Load/store speculation and predication</td>
</tr>
<tr>
<td>Vector IRAM</td>
<td>- UC Berkeley</td>
</tr>
<tr>
<td></td>
<td>- Vector ISA with 8 Kbyte register file</td>
</tr>
<tr>
<td></td>
<td>- Vector memory accesses (strided/indexed)</td>
</tr>
<tr>
<td></td>
<td>- Load speculation and predication</td>
</tr>
<tr>
<td>Impulse</td>
<td>- U. of Utah</td>
</tr>
<tr>
<td></td>
<td>- Programmable memory controller</td>
</tr>
<tr>
<td></td>
<td>- Used for optimized prefetching for regular patterns and remapping for better caching</td>
</tr>
</tbody>
</table>
The Memory Latency Problem

"Moore’s Law"

Processor-Memory Performance Gap: (grows 50% / year)

μProc 60%/yr.

DRAM 7%/yr.
Looking into the Future

- DRAM latency unlikely to be dramatically reduced
- Memory bus latency scales slowly too
- Microprocessors keep improving fast
  - More instructions issued in parallel
  - More datapaths per die
  - More processor cores per die
- Applications are memory-intensive
  - Many media applications do not cache well due to their streaming nature
  - Database applications work on huge data-sets that do not fit in caches

- Memory latency will become even more critical
## Latency Hiding Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger caches</td>
<td>Area; clock cycle; power</td>
</tr>
<tr>
<td>Efficient caches</td>
<td>Clock cycle</td>
</tr>
<tr>
<td>Larger cache blocks</td>
<td>Redundant fetches; power; capacity misses</td>
</tr>
<tr>
<td>HW prefetching</td>
<td>Complexity; redundant fetches; power</td>
</tr>
<tr>
<td>Lock-up free caches</td>
<td>Complexity; requires efficient scheduling</td>
</tr>
<tr>
<td>Speculative loads (dynamic scheduling)</td>
<td>Complexity; redundant fetches; power</td>
</tr>
</tbody>
</table>
Utilizing Memory Bandwidth

- High bandwidth available for future designs
  - multi-bank memory systems, embedded DRAM, cost-efficient MCMs, high-bandwidth interfaces (DDR-II, Rambus)
- How bandwidth is turned into performance
  - **Issue early**: issue memory accesses early
  - **Fetch many**: support a large number of pending accesses to the memory system
  - **Fetch useful**: avoid redundant fetches (less power, less interference)
  - **Utilize access pattern info**: to reduce memory conflicts and maximize caching efficiency
Single-chip Multiprocessors

- Multiple programs/threads run in parallel on multiple CPUs
- Multiplexed access streams utilize the high memory bandwidth
- Each access stream observes the same (long) latency, but the overall system throughput is high
- Very efficient in the presence of coarse-grain parallelism
Multi-threaded Processors

- Multiple threads running on a single core
- When a thread stalls due to memory latency, another thread is executed
- Multiplexed access streams utilize the high memory bandwidth
- Each access stream observes the same (long) latency, but the overall system throughput is high
Vector Architectures

- Multiple memory accesses from one instruction
- Explicit access pattern information
- Decoupling used to tolerate long latencies
- Deep pipelining to amortize the latency over a large number of elements
- Very efficient in the presence of fine-grain parallelism
Summary

- Memory performance enhancement requires the **synergy** of hardware and software techniques
- **Explicitly parallel instruction sets**
  - Provide parallel execution semantics
  - Allow simpler, efficient hardware
  - Enable aggressive optimization opportunities in software
- **Exploiting memory bandwidth in hardware**
  - The remedy to high memory latency
  - Required hardware features:
    - Issue accesses early
    - Support many pending accesses
    - Avoid redundant transfers
    - Utilize access pattern information