

ECE 497NC: Unconventional Computer Architecture

Lecture 2: Processor-In-Memory Architectures 2: Programming and Applications

Outline

- Comparing different processing architectures for PIMs
- An automatic method for assigning program portions to processing resources

Processor Architectures for Active Pages

- Active Pages methodology -- associate some set of functions with each 512 KB “page” of data in memory, use to accelerate performance
- Active Pages implementation -- each “page” on a DRAM has a processing resource associated with it
 - Initial design used reconfigurable logic as the processing resource, had direct connection between processor and DRAM
 - Later studies looked at scalar, VLIW, vector processors, and considered data caches.

Pros/Cons of Different Processor Architectures

- **FPGA**
 - Can customize operations to the needs of each application, which may be of benefit if the application does things that don't match well to a conventional ISA
 - “Compilers” for reconfigurable logic much less advanced than for conventional processors
- **Scalar**
 - Simplest, likely to be smallest, of the processor designs
- **VLIW**
 - Exploit instruction-level parallelism with compiler-scheduled operations
- **Vector**
 - High performance on applications with data parallelism
 - Dense program representation -- one instruction can specify many operations

Architecture Challenges for PIM/Active Pages

- Very limited chip area
 - Many features (out-of-order, etc.) used in current microprocessors won't fit
- Very limited power budget
 - Another reason why they don't use high-area features
 - Low-power constraint changes relative performance of architectures
 - Short pipelines become attractive because of limited clock rate
 - This reduces bypass logic, helping even more

Architectures Considered

- Reconfigurable
 - 256 reconfigurable logic elements per page
- Scalar
 - Single-issue RISC processor
 - Dual-ported instruction cache fetches from both paths of branches
- VLIW
 - 2-8 issue VLIWs considered
 - Support multiple branches/VLIW, up to three
 - Instruction cache fetches from all possible paths
- Vector
 - 4 32-entry vector registers
 - Four functional units execute elements of a vector op in parallel
 - No support for chaining
- Data cache evaluated on base (FPGA) system
 - 512 bytes deemed “sufficient”
 - 4-way set-associativity required to eliminate conflict misses
 - Common result for caches this small

Results

- Scalar architecture always lags others
 - Just not enough processing power
- 4-wide VLIW comparable to FPGA in most cases, noticeably better on one, noticeably worse on one
 - Issue here of how much they did to take advantage of FPGA's strengths
- Adding prefetching boosts VLIW performance over FPGA on half the cases, still one where FPGA wins
- Vector outperformed VLIW on two media applications out of four (different apps. than the other study)
 - Vector lost on MMX because of data cache issues
 - Improvements to vector architecture let it pass VLIW on Median Filter as well.

Automatic Code Mapping for Intelligent Memory

- Model: Powerful host processor, less-powerful memory processor on DRAM chip.
 - For simplicity, assume just one memory chip to avoid issues of partitioning across DRAMs.
- Goal: Automatically (in compiler) divide programs into modules that will run on the different processors. Schedule modules on processors to minimize execution time.
- Approach:
 1. Generate modules (nested loops) such that all the code in a module is likely to have similar behavior.
 2. Determine which processor a module executes more quickly on.
 3. If possible, parallelize module execution across the processors, balancing loads. Otherwise, run module on the processor it does the best on.

Memory Coherence

- Assume that host processor and memory communicate through a standard memory bus
 - No automatic coherence between host processor's and memory processor's caches.
 - Do assume hardware in the intelligent memory sends the most recent value of any location to the host processor on a read.
- Before memory processor can start a module, host processor must write all lines that memory processor will read and any lines that memory processor will only write part of back to memory.
- Before host processor starts a module, it invalidates any lines that the memory processor may have modified.

Where Should Modules Execute?

- A module has *affinity* for the processor where it executes most quickly.
 - Use a combination of static performance modeling and run-time profiling to determine affinities.
- Modules that can't execute in parallel execute on the processor they have affinity for
- Modules that can execute in parallel get distributed across the two processors in a way that balances run-times.

Static vs. Dynamic Partitioning

- Affinity of a module may vary over time
 - Might be function of data
 - Might be function of loop iteration
- When affinity varies, need to dynamically select processor to execute on
- Scheduling
 - Coarse-grained: Schedule each module when it begins executing
 - Fine-grained: Re-evaluate at the start of each outer loop iteration
 - Basic: Time one execution/iteration on each processor, use that to select best from then on
 - Most recent: If execution time of one invocation exceeds the last measured execution time on the other resource, switch
 - First Invocation: Do timing analysis only the first time a module is invoked.

Results

- Dynamic scheduling often helps. Fine-grained scheduling seems to impose too much scheduling overhead
- With good dynamic scheduling and overlapping, numeric applications generally are at least as fast as “ideal” execution, where each module executes sequentially on the processor it does better on with no overheads.
- Non-numeric applications generally perform significantly better than on the worse of the two processors, often close to “ideal”
 - Can’t do better than “ideal” because their compiler can’t overlap execution of non-numeric applications.