Single-Chip Multiprocessors: The Next Wave of Computer Architecture Innovation

Guri Sohi
University of Wisconsin

Also consult for Sun Microsystems
Outline

- Waves of innovation in architecture
- Innovation in uniprocessors
- American football
- Uniprocessor innovation postmortem
- Models for parallel execution
- Future CMP microarchitectures
Waves of Research and Innovation

- A new direction is proposed or new opportunity becomes available
- The center of gravity of the research community shifts to that direction
  - SIMD architectures in the 1960s
  - HLL computer architectures in the 1970s
  - RISC architectures in the early 1980s
  - Shared-memory MPs in the late 1980s
  - OOO speculative execution processors in the 1990s
Waves

- Wave is especially strong when coupled with a “step function” change in technology
  - Integration of a processor on a single chip
  - Integration of a multiprocessor on a chip
Uniprocessor Innovation Wave: Part 1

- Many years of multi-chip implementations
  - Generally microprogrammed control
  - Major research topics: microprogramming, pipelining, quantitative measures
- Significant research in multiprocessors
Uniprocessor Innovation Wave: Part 2

- Integration of processor on a single chip
  - The inflexion point
  - Argued for different architecture (RISC)
- More transistors allow for different models
  - Speculative execution
- Then the rebirth of uniprocessors
  - Continue the journey of innovation
  - Totally rethink uniprocessor microarchitecture
  - Jim Keller: “Golden Age of Microarchitecture”
Uniprocessor Innovation Wave: Results

- Current uniprocessor very different from 1980’s uniprocessor
- Uniprocessor research dominates conferences
- MICRO comes back from the dead
  - Portland was turning point
  - Top 1% (Citeseer)

Source: Hill and Rajwar, 2001
Why Uniprocessor Innovation Wave?

- Innovation needed to happen
  - Alternatives (multiprocessors) not practical option for using additional transistors
- Innovation could happen: things could be done differently
  - Identify barriers (e.g., to performance)
  - Use transistors to overcome barriers (e.g., via speculation)
Lessons from Uniprocessors

- Don’t underestimate potential innovation
- Barriers or limits become opportunities for innovation
  - Via novel forms of speculation
  - E.g., barriers in Wall’s study on limits of ILP
American Football

- Gaining yardage
- Have running game
  - Build entire team and game plan around running
  - Big offensive line
  - Come up with clever running plays
American Football

- Now have a passing game
  - Allows much better yardage gain with a different type of player.
- Coach rethinks entire team with new capability
  - Different offensive line
  - Different plays
- It’s a whole new ball game!!
Uniprocessor Evolution Postmortem

- Improve single program performance
- Uniprocessor was the only option
  - ILP is the parallelism of choice
- Power was not a constraint initially
  - Power inefficiency OK
- Speculation needed to expose ILP and overcome barriers to ILP
  - In-band support for speculation
    - OK for small amounts of speculation
Build big uniprocessor with support for variety of forms of speculation
- Like big offensive line
- Not power efficient, but met then power budget

Can overlap small latencies but longer latencies are problematic
Uniprocessor Evolution Postmortem

- Uniprocessor capabilities underutilized
  - Put more threads on it for multithreaded workloads
  - Increase utilization of (power-inefficient) resources
- Lots of speculation to tolerate large latencies
  - Out-of-band support for speculation
- Free ride for programmers: performance without additional effort
Single Program Performance

- Parallel execution of low-latency operations (if possible)
- Support for speculation
  - To expose parallelism
- Overlap long latency operations with (speculative) computation
- Will need clever ways for above in CMP
Current State

- Inflexion point: can put multiple cores on chip
  - ILP not only option for parallelism
  - Multithreading each core not only option for supporting multiple threads
  - Processor customization for special functions possible

- We now have a passing game
  - How to develop plays for a passing game?
Big Picture CMP Issues

- What should the microarchitecture for a CMP be?
  - Types of cores
  - Memory hierarchies
  - Inter-core communication

- What will be running on the CMP?
  - No free ride for programmers
  - But price of ride can’t be high
Roadmap for CMP Expectations

- Summary of traditional parallel processing
- Revisiting traditional barriers and overheads to parallel processing
- Expectations for CMP applications and workloads
- CMP microarchitectures
Multiprocessor Architecture

- Take state-of-the-art uniprocessor
- Connect several together with a suitable network
  - Have to live with defined interfaces
- Expend hardware to provide cache coherence and streamline inter-node communication
  - Have to live with defined interfaces
Software Responsibilities

- Reason about parallelism, execution times and overheads
  - This is hard
- Use synchronization to ease reasoning
  - Parallel trends towards serial with the use of synchronization
- Very difficult to parallelize transparently
Net Result

- Difficult to get parallelism speedup
  - Computation is serialized
  - Inter-node communication latencies exacerbate problem
- Multiprocessors rarely used for parallel execution
- Typical use: improve throughput
- This will have to change
  - Will need to rethink “parallelization”
Rethinking Parallelization

- Speculative multithreading
- Speculation to overcoming other performance barriers
- Revisiting computation models for parallelization
- How will parallelization be achieved?
Speculative Multithreading

- Speculatively parallelize an application
  - Use speculation to overcome ambiguous dependences
  - Use hardware support to recover from mis-speculation
- E.g., multiscalar
- Use hardware to overcome software barrier to parallelization
Overcoming Barriers: Memory Models

- Weak models proposed to overcome performance limitations of SC
- Speculation used to overcome “maybe” dependences
- Series of papers showing SC can achieve performance of weak models
Implications

- Strong memory models not necessarily low performance
- Programmer does not have to reason about weak models
- More likely to have parallel programs written
Overcoming Barriers: Synchronization

- Synchronization to avoid "maybe" dependences
  - Causes serialization
- Speculate to overcome serialization
- Recent work on techniques to dynamically elide synchronization constructs
Implications

- Programmer can make liberal use of synchronization to ease programming
- Little performance impact of synchronization
- More likely to have parallel programs written
Revisiting Parallelization Models

- **Transactions**
  - simplify writing of parallel code
  - very high overhead to implement semantics in software
- **Hardware support for transactions will exist**
  - Similar to hardware for out-of-band speculation
  - Speculative multithreading is ordered transactions
  - No software overhead to implement semantics
- **More applications likely to be written with transactions**
Other Models: Example 1

```c
for (i=0; i<CHUNKS; i++) {
    ...
    for (j=0; j<CHUNK_SIZE; j++) {
        ...
        random_text[i][j] = ran() * 256;
        ...
    }
    ...
}

hi = seedi/_Q_QUOTIENT;
lo = seedi%_Q_QUOTIENT;
T = _A_MULTIPLIER*lo-_R_REMAINDER*hi;
if (T > 0)
    seedi = T;
else
    seedi = T + _M_MODULUS;
return ( (float) seedi / _M_MODULUS);
```
Example 1
Other Models: Example 2

```c
... route_net (...) {
    ...
    for (...) {
        add_to_heap() {
            alloc ()
            insert element
            compute cost
            heapify ()
        }
        get_heap_head() {
            return top
            fix up heap ()
        }
    } ...
}
```

```c
add_to_heap () {
    alloc ()
    insert element
    compute cost
    heapify ()
}
```

```c
get_heap_head () {
    return top
    fix up heap ()
}
```

```c
alloc (size) {
    lock memheap
    scan memheap
    full to empty
    if free
    obtain
    unlock
    identify bin ~ size
    return
}
```

(175.vpr:route.c)
Example 2

... route() ...
get_heap_head()
alloc()
... route() ...
get_heap_head()
alloc()
... ...

route()
get_heap_head()
Lookup new adds
get_heap_head()
Lookup new adds
route()
get_heap_head()
Lookup new adds
alloc()
alloc()
alloc()
...
Other Expectations for Future Code

- Significant pressure for robust, reliable applications
- Code will have additional functionality for error checking, etc.
  - Overhead code considered perf. barrier
  - Overhead code is parallelization opportunity
- Successful parallelization of overhead will encourage even more use
Example-array bounds checks

- Add two arrays:

```java
void add(int[] a1, int[] a2) {
    for (int i = 0; i < result.length; i++) {
        result[i] = a[i] + b[i];
    }
}
```
Example-array bounds checks

- Loop body
  - w/o checks = 12 insts
  - with checks = 21 insts

<table>
<thead>
<tr>
<th>No array-bounds checks</th>
<th>With array-bounds checks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>sll %o2, 2, %o0</td>
<td>add %i1, %o3, %o7</td>
</tr>
<tr>
<td>cmp %o2, %o5</td>
<td>cmp %o2, %g1</td>
</tr>
<tr>
<td>add %o0, %i1, %o4</td>
<td>bge .LL14</td>
</tr>
<tr>
<td>add %o0, %i0, %o1</td>
<td>add %i0, %o3, %o4</td>
</tr>
<tr>
<td>add %o0, %o7, %o3</td>
<td>ld [%i0+%i0(x)], %o1</td>
</tr>
<tr>
<td>bge .LL12</td>
<td>ld [%o1+8], %o0</td>
</tr>
<tr>
<td>add %o2, 1, %o2</td>
<td>add %o1, %o3, %o5</td>
</tr>
<tr>
<td>ld [%o1+12], %o0</td>
<td>cmp %o2, %o0</td>
</tr>
<tr>
<td>ld [%o4+12], %o1</td>
<td>bgeu .LL23</td>
</tr>
<tr>
<td>add %o0, %o1, %o0</td>
<td>add %o3, 4, %o3</td>
</tr>
<tr>
<td>b .LL13</td>
<td>cmp %o2, %g1</td>
</tr>
<tr>
<td>st %o0, [%o3+12]</td>
<td>bgeu .LL23</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td>ld [%i1+8], %o0</td>
</tr>
<tr>
<td></td>
<td>cmp %o2, %o0</td>
</tr>
<tr>
<td></td>
<td>bgeu .LL23</td>
</tr>
<tr>
<td></td>
<td>ld [%o4+4], %o1</td>
</tr>
<tr>
<td></td>
<td>ld [%o7+4], %o0</td>
</tr>
<tr>
<td></td>
<td>add %o2, 1, %o2</td>
</tr>
<tr>
<td></td>
<td>add %o1, %o0, %o0</td>
</tr>
<tr>
<td></td>
<td>b .LL15</td>
</tr>
<tr>
<td></td>
<td>st %o0, [%o5+4]</td>
</tr>
</tbody>
</table>
Divide program into tasks

Execute checking code in parallel with task

Checking code commits or aborts task computation
Other software requirements

- Robust, error-tolerant software will be needed to work on error-prone hardware
  - Likely abundant source of parallelism
  - Need to figure out how to do this
Vehicle for Parallelization

- Tradition: automatic or manual parallelization of user-level code
  - Too much code to target
  - Too difficult
- Successful vehicle for parallelization will target code selectively
- OS and libraries
  - Written to facilitate parallel execution
Impact of Parallelization

- Expect different characteristics for code on each core
- More reliance on inter-core parallelism
- Less reliance on intra-core parallelism
- May have specialized cores
Microarchitectural Implications

- Processor Cores
  - Skinny, less complex
  - Perhaps specialized
- Memory structures (i-caches, TLBs, d-caches)
  - Significant sharing possible
  - Low-overhead communication possible
Microarchitectural Implications

- Novel Coherence Techniques (e.g., separate performance from correctness
  - Token coherence
  - Coherence decoupling
- Pressure on non-core techniques to tolerate longer latencies
  - Helper threads, pre-execution
  - Other novel memory hierarchy techniques
Microarchitectural Implications

- Significant increase in bandwidth demand
  - Use on-chip resources to attenuate off-chip bandwidth demand
Summary

- It’s a whole new ball game!
- New opportunities for innovation in MPs
- New opportunities for parallelizing applications
- Expect little resemblance between MPs today and CMPs in 15 years
- We need to invent and define differences