μ-op Fission: Hyper-threading without the Hyper-headache

Anthony Cartolano
Robert Koutsoyannis
Daniel S. McFarlin
Carnegie Mellon University
Dept. of Electrical and Computer Engineering
Streaming WHT Results on Intel Multicore: Best Versus Single Core

Penryn, Core i7 Best vs. Single Core Performance

performance [Gflop/s]
Characterized by degree of sharing $\alpha$ (Cost of Communication)$^{-1}$

- SIMD: share register file, functional units, cache hierarchy
- SMP: share main memory and QPI link
Hierarchy of Software Interfaces?

- General Purpose SW threading model to rule them all: affinity knob
- Overhead: intrinsics vs. OpenMP vs. APIs + Runtimes
- Amortizing runtime overhead dictates partitioning: precludes SMT
View SMT and SIMD as a continuum for parallel numeric code

Expose SMT to SW and Encode SMT loop parallelization with intrinsics to minimize overhead

Requires HW/SW support and co-optimization to achieve
Outline

- Motivation
- SMT Hardware Bottlenecks
- μ-op Fission
- Required software support
- Experimental Results
- Concluding Remarks
Generic N-way SMT

- Narrow Front-End vs. Wide Backend
2-Way SMT Case Study

- Front-End (Fetch & Decode) is implemented as Fine-Grained Multi-threading
  - Done to reduce i-cache, decode latency
  - Can lead to back-end starvation
μ-op Fission

- 2 iterations compiler-fused with clone bit
  - Equivalent to unroll-and-jam optimization
  - Decoder fissions iterations into available RATs
  - Reduces fetch & decode bandwidth and power
  - Allows narrow front-end to keep wide back-end pipeline full
Formal HW/SW Co-Design: The Data Pump Architecture

- SW Parameterizable
- NUMA
- SW Programmable Memory Controller
- Decoupled Compute & Communication Instruction Streams
- Asynch Gather/Scatter
  - MEM <-> LM
  - VRF <-> LM
- Simple Manycore
- SIMD
Software Architecture: Spiral

- Library generator for linear transforms (DFT, DCT, DWT, filters, ....) and recently more ...

- Wide range of platforms supported: scalar, fixed point, vector, parallel, Verilog, GPU

- Research Goal: “Teach” computers to write fast libraries
  - Complete automation of implementation and optimization
  - Conquer the “high” algorithm level for automation

- When a new platform comes out: Regenerate a retuned library

- When a new platform paradigm comes out (e.g., CPU+GPU): Update the tool rather than rewriting the library

Intel uses Spiral to generate parts of their MKL and IPP libraries
Spiral: A Domain Specific Program Generator

Transform
user specified

Fast algorithm
in SPL
many choices

Optimization at all
abstraction levels

parallelization
vectorization

loop
optimizations

constant folding
scheduling
......

Iteration of this process to search for the fastest
SAR_{k \times m \rightarrow n \times n} \rightarrow \text{DFT}_{n \times n} \circ \text{Interp}_{k \times m \rightarrow n \times n}

\text{DFT}_{n \times n} \rightarrow (\text{DFT}_{n} \otimes I_{n}) \circ (I_{n} \otimes \text{DFT}_{n})

\text{Interp}_{k \times m \rightarrow n \times n} \rightarrow (\text{Interp}_{k \rightarrow n} \otimes I_{n}) \circ (I_{k} \otimes I_{n} \otimes \text{Interp}_{m \rightarrow n})

\text{Interp}_{r \rightarrow s} \rightarrow \left( \bigoplus_{i=0}^{n-2} \text{InterpSeg}_{k} \right) \oplus \text{InterpSegPruned}_{k, \ell}

\text{InterpSeg}_{k} \rightarrow G_{f}^{u \cdot n \rightarrow k} \circ \text{iPrunedDFT}_{n \rightarrow u \cdot n} \circ \left( \frac{1}{n} \right) \circ \text{DFT}_{n}
Spiral’s Automatically Generated PFA SAR Image Formation Code

SAR Image Formation on Intel platforms
performance [Gflop/s]

- Algorithm by J. Rudin (best paper award, HPEC 2007): 30 Gflop/s on Cell
- Each implementation: vectorized, threaded, cache tuned, ~13 MB of code
- Code was not written by a human
Required Software Support

- Executable has stub code which initializes pagetables and other CPU control registers at load time on all HW contexts

- Compiler performs virtual Loop-Unroll-and-Jam on tagged loops
  - Maximizes sharing
  - SMT Thread Cyclic Partitioning

```
i=0: A;B;C;D;
i=1: A;B;C;D;
i=2: A;B;C;D;
i=3: A;B;C;D;
```

```
i=0: A;B;C;D;
i=1: A;B;C;D;
i=2: A;B;C;D;
i=3: A;B;C;D;
```

```
i=0: A;B;C;D;
i=1: A;B;C;D;
i=2: A;B;C;D;
i=3: A;B;C;D;
```

```
i=0: A;B;C;D;
i=1: A;B;C;D;
i=2: A;B;C;D;
i=3: A;B;C;D;
```

```
i=0: A;B;C;D;
i=1: A;B;C;D;
i=2: A;B;C;D;
i=3: A;B;C;D;
```

```
th0 th1 th2 th3
```
Fork Support

- Need a lightweight fork mechanism
  - Presently, can only communicate between SMT register files via memory

### Compile Time

- Store:
  - `store 0, (a0,0);`
  - `store 1, (a0,1);`
  - `store 2, (a0,2);`
  - `store 3, (a0,3);`
- Load:
  - `load (a0, APIC_ID), i;`
  - `load (a0,0), i;`
  - `load (a0,1), i;`
  - `load (a0,2), i;`
  - `load (a0,3), i;`

- Load/Store Queue prevents materialization in most cases

- Prefer to have multi-assign statement for loop index with a vector input
  - `i = {0,1,2,3};` → `i = 0; i = 1; i = 2; i = 3;`

- Need broadcast assignment for live-in set to the loop
  - `load addr, a0;` → `load addr, a0;` → `load addr, a0;`
Sample Loop Execution

### Compile Time

- i = {0,1};
- load addr, a0;
- L0: cmp i, n;
- jmpgte END;
- A;B;C;D;
- add 2, i;
- jmp L0

END: waitrobs

- i = 0;
- load addr, a0;
- L0: cmp i, n;
- jmpgte END;
- A;B;C;D;
- add 2, i;
- jmp L0

END: waitrobs

- i = 1;
- load addr, a0;
- L0: cmp i, n;
- flushgte
- A;B;C;D;
- add 2, i;

END: waitrobs
Experimental Setup

- Used PTLSim with above configuration
  - Larger ROB size + physical register size than Nehalem
  - Smaller number of functional units
  - Simulate μ-op fission with explicit unroll-and-jam of source code coupled with penalized functional unit latencies
Experimental Results

Speedup of Various Kernels via $\mu$-op Fission SMT
percentage reduction in total cycle count

- SSE Interpolation
- Scalar Interpolation
- Streaming WHT
- Non-2 Power DFTs
- Binary Matrix Factorization

Legend:
- 2-way SMT
- 4-way SMT
Results Drilldown

Performance Improvement of Various μArch Metrics for μ-op Fissioned SSE interpolation Kernel

Percentage improvement over baseline

- 2-way SMT
- 4-way SMT

<table>
<thead>
<tr>
<th>Metric</th>
<th>2-way SMT</th>
<th>4-way SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads/cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Insns/cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int Insns/cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROB Hit Rate/result/cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replays</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Concluding Remarks

- Demonstrated a HW/SW Co-optimization approach to SMT parallelization

- Preliminary evaluation suggests performance benefit for a range of numerical kernels

- Scales with number of SMT contexts

- “Thread Fusion” research suggests a 10-15% power consumption reduction is possible due to reduced fetch/decode

- Future work: handling control-flow with predication and diverge-merge
THANK YOU!

Questions?