Revisiting the Sequential Programming Model for Multi-Core
Matthew J. Bridges Neil Vachharajani Yun Zhang Thomas Jablin David I. August
Department of Computer Science
Princeton University

Presenter: Kishen Maloor
Dept of Computer & Information Sciences
University of Delaware

CISC 879 : Software Support for Multicore Architectures
Motivation

- Move to multi-threaded programming is costly.
  - Parallel programming models: costly to adopt.

- Need for automatic parallelization.
  - Large number of existing single-threaded applications.

- Past attempts have been Insufficient to keep many cores busy.
• **Argument:** Not a limitation of the sequential programming model.

• **How can we fix things?**
  - Build a framework for automatic thread extraction.
  - Tweak the sequential programming model a small bit.
Parallelization Framework

• Compiler and hardware support.
  - Thread level speculation.
    - Execute loop iterations in parallel.
  - Needs to buffer results.
  - Decoupled software pipelining.
    - Partition loop into stages; execute in parallel.
Parallelization Framework

- Attempt to extract DOALL parallelism.
- Use of alias and value speculation.
- Avoid misspeculation.
  - Synchronizing some dependences.
  - Forwarding stored values to later threads.
• Compilation Scope.
  - Significant parallelism exists in the outermost loop.
  - Need to leverage parallelism at any loop level.
  - Non-trivial problem.
  - Use of whole program optimization.

• Extend sequential programming model
  - Y-branch
  - Commutative
Use of a y-branch

```c
#define CUTOFF 100000

int count = 0;

while ((char = read(1)) != EOF) {
    profitable = compress(char, dict)
    if (!profitable) {
        dict = restart_dictionary(dict);
    } else if (count == CUTOFF) {
        dict = restart_dictionary(dict);
        count = 0;
    }
    count++;
}

finish_dictionary(dict);
```
static int seed;

@Commutative
int Yacm_random() {
    int temp = seed / 127773L;
    seed = 16807L * (seed - temp * 127773L) - (temp * 2836L);
    if( seed < 0 )
        seed += 2147483647L;
    Return seed;
}

Use of commutative
Experimentation Approach

Construct an application execution plan.

- Decompose loops into 3 phases.
- Phase 1 tasks depend on prior phase 1 tasks.
- Phase 2 tasks depend on corresponding phase 1 task.
- Phase 3 tasks depend on corresponding phase 2 task and prior phase 3 tasks.
while (condition) {
    A: line = read();
    B: result = work(line);
    C: printf(result);
}

(a) Example Code

(b) Static Stage Dependences

(c) Potential Task Execution

CISC 879 : Software Support for Multicore Architectures
• Extension of DSWP.
  - Uses speculation

• Phases are statically selected “regions” in code.

• “Tasks” are dynamic instances of these regions.
Experimentation Approach

• Task dependence used to model speculation.

• Instrument code to use hardware performance counters.

• Code compiled using GCC (-O3)

• Simulator reports total parallel execution time.
Case Studies

• Manual parallelization of the SPEC CINT2000 benchmarks.

• Uses the described experimentation approach.

• Demonstrate use of known compiler technologies.

• Experiments performed with 1-32 cores.
• Compresses and decompresses a file.

• Input file is divided into independent blocks of same size.

• Use DSWP parallelization.
  - Phase A thread reads in blocks.
  - Phase B threads compress blocks, buffer results.
  - Phase C threads write result to output stream.
• Object oriented database transaction benchmark.

• Loop in DB Test function calls Lookup, Delete, and Create functions.

• Create and Delete functions executed in parallel.

• STATUS variable speculated to be NORMAL.

• Alias speculation to handle DB structure update dependencies.
253.perlbmk

- Interpreter for the Perl language.
- Source statements => set of operations demarcated by NEXTSTATE operations.
- Executed using a virtual stack machine.
- Compiler can precompute next NEXTSTATE operation.
- Execute sets of operations representing perl statements in parallel.
181.mcf

- Solves a combinatorial optimization problem using a network simplex algorithm.
- Main loop of this algorithm parallelized using value speculation.
176.gcc

- Compiles C programs to MIPS assembly.
- Compiler optimization sequence dominates runtime.
- Optimization sequence can be run in parallel for functions.
- Symbol table lookup and insert functions marked "commutative".
- Some supporting functions rewritten.
• Interpreter for a computational discrete algebra programming language.

• Speculate that statements are data independent.

• Memory allocation routines marked commutative.

• Misspeculation results:
  - Due to true data dependences.
  - Due the garbage collection performed.
• Application that plays chess.
• Uses a recursive search function.
• Can search each of the moves in the root list of moves independently.
• Uses caches to some ways to prune the search space and improve performance.
• Cache lookup function marked as commutative.
• Unroll recursion and parallelize.
• Parses sentences, check if grammatically correct.
• Parse function dominates the runtime.
• Parsing sentences done in parallel.
• Memory allocator marked commutative.
• No garbage collection.
Perform place and route simulation.

- Value and alias speculation used.
- Alias violation on the block and network structures – Miss-speculation.
- Random number generator be marked as Commutative.
• FPGA place and route calculations.
• Pseudo-random number generator is used to choose a block and position to move/swap blocks – marked commutative.
• Swap function parallelized.
• Value speculation: No change in loads of block coordinates, network structures.
• Requires many threads for performance.
• Compression and decompression.

• Files are compressed in blocks; hard to predict beginning of new block.

• Use of fixed-size blocks reduces compression ratio.

• Solution: Use y-branch – loss of compression only when parallelization achieved.

• When there are many processors, compression loss < 1%.
## Results and Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Threads</th>
<th>Speedup</th>
<th>Moore’s Speedup</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.Gzip</td>
<td>32</td>
<td>29.91</td>
<td>5.38</td>
<td>5.56</td>
</tr>
<tr>
<td>175.Vpr</td>
<td>15</td>
<td>3.59</td>
<td>3.71</td>
<td>0.97</td>
</tr>
<tr>
<td>176.Gcc</td>
<td>16</td>
<td>5.06</td>
<td>3.84</td>
<td>1.32</td>
</tr>
<tr>
<td>181.Mcf</td>
<td>32</td>
<td>2.84</td>
<td>5.38</td>
<td>0.53</td>
</tr>
<tr>
<td>186.Crafty</td>
<td>32</td>
<td>25.18</td>
<td>5.38</td>
<td>4.68</td>
</tr>
<tr>
<td>197.parser</td>
<td>32</td>
<td>24.50</td>
<td>5.38</td>
<td>4.55</td>
</tr>
<tr>
<td>253.Perlbmk</td>
<td>5</td>
<td>1.21</td>
<td>2.18</td>
<td>0.55</td>
</tr>
<tr>
<td>254.gap</td>
<td>10</td>
<td>1.94</td>
<td>3.05</td>
<td>0.64</td>
</tr>
<tr>
<td>255.vortex</td>
<td>32</td>
<td>4.92</td>
<td>5.38</td>
<td>0.91</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>12</td>
<td>6.72</td>
<td>3.34</td>
<td>2.01</td>
</tr>
<tr>
<td>300.twolf</td>
<td>8</td>
<td>2.06</td>
<td>2.74</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Moore’s law speedup: doubling of cores yields 1.4x speedup.
Summary and Conclusion

- Case study results demonstrate parallelization opportunities.

- Conclusion: Using the right combination of available technologies, automated parallelization is possible!
Questions?

Thanks for your time…