How to Make Loop-level Data Dependence Profiling Fast and Memory Efficient?

Hongtao Yu, Zhiyuan Li

Department of Computer Science
Purdue University
The bigger picture

- How to retrofit existing application software for multicore processors?
  - Difficulties with programmer writing parallel programs
    - Race conditions
    - Memory consistency model
    - Execution model (process, thread, stream, vector, …)
    - Granularity
  - Difficulties with using compilers to help
    - Alias problems
    - Symbolic analysis
    - Input-dependent behaviors
    - etc
In both cases, a key issue is to understand the inherent data dependences thoroughly.
Loop-level data dependence

while (...) {
    ...
    for (i = 0; i <= 65536; i++)
        ftab[i] = 0;
    c1 = block[-1];
    for (i = 0; i <= last; i++) {
        c2 = block[i];
        ftab[(c1 << 8) + c2]++; c1 = c2;
    }
    for (i = 1; i <= 65536; i++)
        ftab[i] += ftab[i-1];
    c1 = block[0];
    for (i = 0; i < last; i++) {
        c2 = block[i+1];
        j = (c1 << 8) + c2;
        c1 = c2;
        ftab[j]--; zptr[ftab[j]] = i;
    }
    ...
    ...
    for () {
        ...
        ... = zptr[..]
    } /* end while */

• Zptr[] and ftab[] are both privatizable;
• No flow dependence actually exists across the while-loop iterations;
• While loop is a doacross kind of loop w/ significant parallelism
• Current compilers cannot determine yet
Data dependence profiling can be an important tool for such understanding:

- Compiler writers can discover areas for improvement
- Programmers can find code segments to rewrite
- With speculative parallel execution support, we’ll know which loop to speculate

Unfortunately, when done in a straightforward way, data dependence profiling is extremely time consuming:

- Which is perhaps why it has not been widely used in practice

We look for ways to make it practical:

- Reduce memory use (otherwise profiling may not even be feasible)
- Reduce time (by one or two orders of magnitude)
Techniques to discuss today

- **Basic mechanisms**
  - Hash tables for memory addresses
  - Regular strides (and limitation)

- **Obvious compiler-based techniques to reduce cost**
  - Some works but some don’t

- **“Multi-slicing” (or parallel profiling based on partitioning)**
  - Partitioning by alias classes
  - Partitioning by potential dependence edges
  - Input-dependent specialization (“thinned analysis”)

- **To make profiling much more practical**
  - Partial profiling that is sufficient for loop parallelization

- **A more efficient mechanism (memory-tags)**
  - Need compiler support
Basic Mechanisms

- Fundamentally there are two ways to check dependences during profiling
  - Memory-address as the base
  - Statement (or load/store instruction) as the base

Check for new reference
Regular strides

- **Method called SD3**
  - When it works, it can speed up profiling by 100x

- **Unfortunately, when loop levels increase**
  - Strides are often not regular anymore

- **When aliasing relationship is unclear**
  - It is unsafe to use strides

- **Fall back to hash tables**

```
<base addr,, stride, upper bound, lower bound  r/w>
```

Check for new reference
Compiler-based improvement

- Obviously, if the dependence question for a pair of load/store operations can be answered by the compiler,
  - then it does not need to be checked by profiling
- However, under the (memory-address) hash-table mechanism
  - If a load/store operation has a possible dependence with any other operation,
    - then it must be registered at (and checked against) the hash table
- Therefore in C programs, we often end up with registering and checking every memory reference
- Nonetheless, there exists a simple opportunity for improvement
Equivalence class and its representative for profiling

- Need symbolic analysis to make sure the address is the same in every iteration
- Need control flow information to identify upward-exposed use

A code segment from 456.hmmer

```c
133:    for (k = 1; k <= M; k++) {
134:       mc[k] = mpp[k-1] + tpmm[k-1];
135:       if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
136:       if ((sc = dp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
137:       if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
138:       mc[k] = mc[k] + ms[k];
139:       if (mc[k] < -INFTY) mc[k] = -INFTY;
140:    
141:    dc[k] = dc[k-1] + tpdd[k-1];
142:    if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
143:    if ((dc[k] < -INFTY) dc[k] = -INFTY;
144:    
145:    if (k < M) {
146:       ic[k] = mpp[k] + tpml[k];
147:       if ((sc = ip[k] + tpil[k]) > ic[k]) ic[k] = sc;
148:       ic[k] += is[k];
149:       if (ic[k] < -INFTY) ic[k] = -INFTY;
150:    };
151: }
```
### Static count of dependence edges

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Dependence checking</th>
<th>Equivalence Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>498202</td>
<td>294665</td>
<td>43344</td>
</tr>
<tr>
<td>429.mcf</td>
<td>12346</td>
<td>9478</td>
<td>844</td>
</tr>
<tr>
<td>433.milc</td>
<td>126336</td>
<td>93711</td>
<td>7358</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>632792</td>
<td>487546</td>
<td>71202</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>5288671</td>
<td>4057566</td>
<td>482911</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>62163</td>
<td>30113</td>
<td>2826</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>5097323</td>
<td>4588307</td>
<td>64744</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>101688</td>
<td>61969</td>
<td>9278</td>
</tr>
</tbody>
</table>
Parallel profiling

- **Basic idea**
  - Instrumentation for a subset of potentially dependent memory references
    - Run instrumentation and profiling for different subsets independently
    - Merge the results to produce the final dependence graph
  - **Overhead due to extra instrumentation and program execution**
    - Controlling granularity and scheduling carefully for load balance
- **Partitioning by alias classes**
- **Partitioning by potential dependence edges**
  - Grouping dependence edges that must be profiled together (to recognize loop-independent dependences)
  - Granularity controlled by available computing resource
  - Dynamic scheduling by work stealing
A high-level overview of the multi-slicing approach
Details of worker processes
Implementation

- In the GCC compiler, version 4.6.
- Experiments with SPECint 2006 benchmark suite
- “Test” size input
Visualizing DD graph by DDGrapher
## Effect of thinned analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Ordinary Analysis</th>
<th>Thinned Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACP   #Edge class</td>
<td>ACP   #Edge class</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>17.7  25661</td>
<td>13.9  21752</td>
</tr>
<tr>
<td>429.mcf</td>
<td>7.3    439</td>
<td>7.2    284</td>
</tr>
<tr>
<td>433.milc</td>
<td>7.4    3967</td>
<td>5.7    3415</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>4.4    49129</td>
<td>1.4    1487</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>10.1   209034</td>
<td>10.0   172156</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>4.2  515</td>
<td>2.9    229</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>6.6    17975</td>
<td>5.9    17975</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>4.8    5336</td>
<td>1.3    3045</td>
</tr>
</tbody>
</table>

ACP (Alias Class Population): the average size of an alias class, counted by the number of load/stores.
## Memory usage during profiling (MB)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>SD³</th>
<th>Max Alias</th>
<th>Max Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>24</td>
<td>3.8G</td>
<td>532</td>
<td>70</td>
</tr>
<tr>
<td>429.mcf</td>
<td>154</td>
<td>3.6G</td>
<td>491</td>
<td>170</td>
</tr>
<tr>
<td>433.milc</td>
<td>9.3</td>
<td>&gt;3G</td>
<td>&gt;337</td>
<td>14</td>
</tr>
<tr>
<td>456.hmmr</td>
<td>1.1</td>
<td>3G</td>
<td>170</td>
<td>53</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>174</td>
<td>&gt;28G</td>
<td>&gt;5.5G</td>
<td>191</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>1.4</td>
<td>&gt;721</td>
<td>223</td>
<td>14</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>26</td>
<td>&gt;1.1G</td>
<td>&gt;352</td>
<td>179</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>30</td>
<td>&gt;1.7G</td>
<td>&gt;301</td>
<td>94</td>
</tr>
</tbody>
</table>
Profiling time with the multi-slicing approach

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>SD³</th>
<th>Alias Class</th>
<th>Edge Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>3.7s</td>
<td>180h</td>
<td>77.8h</td>
<td>21.7h</td>
</tr>
<tr>
<td>429.mcf</td>
<td>6.7s</td>
<td>957h</td>
<td>489.9h</td>
<td>488.5h</td>
</tr>
<tr>
<td>433.milc</td>
<td>19s</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>456.hammer</td>
<td>16.5s</td>
<td>142h</td>
<td>12.8h</td>
<td>5.3h</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>11s</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>7.6s</td>
<td>×</td>
<td>28.4h</td>
<td>16.8h</td>
</tr>
<tr>
<td>464.l264ref</td>
<td>22.6s</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>6.5s</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

All versions compiled with “O0” option.
For profiling, “O0” allows precise drawing of dependence graph at source level
Multi-slicing parallel efficiency (alias-class partitioning)

![Graph showing speedup vs number of cores for different benchmarks.]

- 401.bzip2
- 429.mcf
- 456.hmmer
- 462.libquantum

Number of cores: 1, 4, 8, 16, 32

Speedup: 0, 1, 2, 3, 4, 5, 6, 7, 8
Multi-slicing parallel efficiency (edge partitioning)

![Graph showing speedup vs. number of cores for different benchmarks](image)

- 401.bzip2
- 429.mcf
- 433.milc
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 482.sphinx3

Intel Review Seminar (Santa Clara, CA)
Characteristics of the Benchmarks (“reference” input size)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLOC</th>
<th>File</th>
<th>Loop</th>
<th>Loop profiled</th>
<th>#Run</th>
<th>#Iter</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.prlbench</td>
<td>170</td>
<td>126</td>
<td>1361</td>
<td>perl.c, 4751:4833</td>
<td>270</td>
<td>1.96</td>
<td>1%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>8</td>
<td>12</td>
<td>234</td>
<td>spec.c, 329:377</td>
<td>1</td>
<td>3</td>
<td>99.71%</td>
</tr>
<tr>
<td>403.gcc</td>
<td>521</td>
<td>278</td>
<td>4621</td>
<td>c-parse.c, 1874:4510</td>
<td>1</td>
<td>609709</td>
<td>99.80%</td>
</tr>
<tr>
<td>429.mcf</td>
<td>3</td>
<td>25</td>
<td>53</td>
<td>mcf.c, 48:104</td>
<td>1</td>
<td>6</td>
<td>99.99%</td>
</tr>
<tr>
<td>433.milo</td>
<td>15</td>
<td>86</td>
<td>445</td>
<td>update.c, 40:94</td>
<td>2</td>
<td>2</td>
<td>90.82%</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>197</td>
<td>96</td>
<td>1359</td>
<td>interface/gtp.c, 82:135</td>
<td>1</td>
<td>484</td>
<td>99.98%</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>36</td>
<td>72</td>
<td>897</td>
<td>hmmcalibrate.c, 499:518</td>
<td>1</td>
<td>500000</td>
<td>99.99%</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>14</td>
<td>25</td>
<td>306</td>
<td>epd.c, 313:554</td>
<td>1</td>
<td>9</td>
<td>99.99%</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>4</td>
<td>31</td>
<td>102</td>
<td>expn.c, 45:54</td>
<td>1</td>
<td>21</td>
<td>97.81%</td>
</tr>
<tr>
<td>484.h264ref</td>
<td>51</td>
<td>81</td>
<td>1963</td>
<td>lenenc.c, 305:424</td>
<td>1</td>
<td>62</td>
<td>99.95%</td>
</tr>
<tr>
<td>470.lbm</td>
<td>1.3</td>
<td>6</td>
<td>23</td>
<td>main.c, 36:51</td>
<td>1</td>
<td>300</td>
<td>99.16%</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>25</td>
<td>94</td>
<td>599</td>
<td>spec_main_live_pretend.c:167,189</td>
<td>1</td>
<td>24</td>
<td>99.81%</td>
</tr>
</tbody>
</table>

All versions compiled with “O0” option.
For profiling, “O0” allows precise drawing of dependence graph at source level.
Timing and memory usage result of different profiling techniques ("reference" input size)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (Minutes)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Native</td>
<td>SD$^3$</td>
</tr>
<tr>
<td>400.perlbm</td>
<td>13.26</td>
<td>34.15</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>9.16</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>403.gcc</td>
<td>4.97</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>429.mcf</td>
<td>18.31</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>433.milc</td>
<td>48.22</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>4.46</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>43.83</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>39.72</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>45.71</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>7.84</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>470.lbm</td>
<td>2.88</td>
<td>&gt;15000</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>86.72</td>
<td>&gt;15000</td>
</tr>
</tbody>
</table>

“Thinned analysis” and “partial profile” are both implemented in the “Hash” and “Tag” versions.
Parallel profiling time (minutes) with the memory-tag approach

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Serial</th>
<th>2-core</th>
<th>4-core</th>
<th>8-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>15.48</td>
<td>15.38</td>
<td>15.22</td>
<td>15.26</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>319.65</td>
<td>231.6</td>
<td>221.07</td>
<td>210.22</td>
</tr>
<tr>
<td>429.mcf</td>
<td>203.43</td>
<td>133.40</td>
<td>108.26</td>
<td>89.9</td>
</tr>
<tr>
<td>433.milc</td>
<td>331.72</td>
<td>191.31</td>
<td>163.61</td>
<td>151.68</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>33.49</td>
<td>21.55</td>
<td>15.64</td>
<td>14.46</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>829.61</td>
<td>424.67</td>
<td>264.88</td>
<td>175.41</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>337.64</td>
<td>260.18</td>
<td>240.89</td>
<td>233.61</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>804.93</td>
<td>612.90</td>
<td>576.39</td>
<td>550.26</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>88.54</td>
<td>45.07</td>
<td>24.08</td>
<td>18.64</td>
</tr>
<tr>
<td>470.lbm</td>
<td>43.38</td>
<td>26.41</td>
<td>17.99</td>
<td>16.91</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>564.38</td>
<td>368.87</td>
<td>231.89</td>
<td>212.40</td>
</tr>
</tbody>
</table>
Idea of partial but sufficient profiling

- A partial dependence graph
  - contains only the latest data dependence concerned with each statement
  - The access record for each monitored memory address keeps only the most recent load and store.
- Under such simplification, we are able to capture the most recent instance of loop-independent and loop-carried dependence.
- We also keep the shortest dependence distance
Algorithm 1 Partially Check Address.

1: procedure check_address(addr, opid, kind)
2: begin
3: if erno > 0 then
4:  \( < id_1, ierno_1, kind_1 > = hash[addr].last_read \)
5:  \( < id_2, ierno_2, kind_2 > = hash[addr].last_write \)
6: check whether \( < addr, opid, kind > \) depends on either/both of the above triplets
7:  if kind=="read" then
8:  hash[addr].last_read = \( < opid, ierno, kind > \)
9:  else
10:  hash[addr].last_write = \( < opid, ierno, kind > \)
11:  end if
12: end if
13: end
Idea of partial but sufficient profiling

- **Theorem 1 (Fundamental Theorem of Dependence):**
  
  "Any reordering transformation that preserves every dependence in a program preserves the meaning of that program."

- **Theorem 2:**
  
  A transformation or a loop execution schedule violates a dependence in the partial dependence graph if and only if it violates a dependence in the complete dependence graph.
Memory tagging

- **Avoid hash table operations**
  - Especially to avoid byte-level hashing

- **Embed memory tags in the original data structure**
  - Direct comparison between memory tags
  - Fixed offset from the base address

- **Requires careful type analysis**
  - Especially in the presence of type cast, union, aliasing

- **Look for “promotable” objects**
  - Set up promotion rules
Recap

- **Basic mechanisms**
  - Hash tables for memory addresses
  - Regular strides (and limitation)

- **Obvious compiler-based techniques to reduce cost**
  - Some works but some don’t

- **“Multi-slicing” (or parallel profiling based on partitioning)**
  - Partitioning by alias classes
  - Partitioning by potential dependence edges
  - Input-dependent specialization (“thinned analysis”)

- **To make profiling much more practical**
  - Partial profiling that is sufficient for loop parallelization

- **A more efficient mechanism (memory-tags)**
  - Need compiler support

- **Future work:**
  - We are yet to combine partial profiling and memory tagging with edge partitioning
  - Use profiling result to produce parallel code automatically
To be continued …