Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization

Alain Ketterlin Philippe Clauss
Motivation

- Data dependence is central for:
  - parallelization
  - locality optimization
  - ...

- Compilers have limited capabilities
  - aliasing
  - fine grain

- Parwiz: an empirical approach
  - uses dynamic information
  - targeting fine or coarse-grain parallelism
  - includes several decision/parallelization algorithms
  - leaves final validation to the programmer
Data-dependence

- For every access to address \( a \)
  - What was the previous access to \( a \)?
  - A *shadow memory* tracks last accesses

Program structures

- Program execution is a hierarchy of calls and loops
- Correlate accesses (and dependencies) with calls and loops
- An *execution point* uniquely locates every access

\[
\text{call} \leftarrow p_0 \quad \text{loop} \leftarrow i_0 \quad \text{iter} \leftarrow p_1 \quad \text{call} \leftarrow p_3 \quad \text{loop} \leftarrow i_1 \quad \text{iter} \leftarrow p_2 \quad \text{access}
\]

(carries a generalized iteration vector)
An execution tree keeps “all” execution points (a dynamic call tree, plus nodes for loops and iterations).

A dependence is carried by the lowest common ancestor on both execution points.

A dependence domain may span several levels of the tree.
Example:

- (17)–(42)
Example:

- (17)–(42)
- (17, 0) – (42, 68) and (42, 68) – (42, 91)
Framework > Algorithm: Parwiz

Execution tree

Dep. table \( \#n \)

\[
\text{Shadow Mem.} \\
\begin{array}{|c|c|}
\hline
0xabcd & x_o \\
\hline
\end{array}
\]
Framework > Implementation

- Tool architecture

  - Static Analyzer
  - Instrumented Program
  - Dependence Profiler

  ![Diagram](image)

- Static analyzer: computes CFG and loop hierarchies

- Instrumentation
  - function call/return
  - loop entry/iteration/exit
  - memory accesses

- Works from x86_64 code, requires no compiler support

- Instrumentation/tracing done with Pin
Applications > Loop parallelism (1)

- all loops from the SPEC OMP-2001 programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Loops</td>
</tr>
<tr>
<td>312.swim_m</td>
<td>26</td>
</tr>
<tr>
<td>314.mgrid_m</td>
<td>58</td>
</tr>
<tr>
<td>316.applu_m</td>
<td>168</td>
</tr>
<tr>
<td>318.galgel_m</td>
<td>541</td>
</tr>
<tr>
<td>320.equake_m</td>
<td>73</td>
</tr>
<tr>
<td>324.apsi_m</td>
<td>191</td>
</tr>
<tr>
<td>326.gafort_m</td>
<td>58</td>
</tr>
<tr>
<td>328.fma3d_m</td>
<td>233</td>
</tr>
<tr>
<td>330.art_m</td>
<td>79</td>
</tr>
<tr>
<td>332.ammp_m</td>
<td>76</td>
</tr>
</tbody>
</table>
all loops from the SPEC OMP-2001 programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Executed</th>
<th>Slowdown/overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Loops</td>
<td>#Par.</td>
</tr>
<tr>
<td>312.swim_m</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>314.mgrid_m</td>
<td>58</td>
<td>52</td>
</tr>
<tr>
<td>316.applu_m</td>
<td>168</td>
<td>135</td>
</tr>
<tr>
<td>318.galgel_m</td>
<td>541</td>
<td>455</td>
</tr>
<tr>
<td>320.equake_m</td>
<td>73</td>
<td>67</td>
</tr>
<tr>
<td>324.apsi_m</td>
<td>191</td>
<td>147</td>
</tr>
<tr>
<td>326.gafort_m</td>
<td>58</td>
<td>43</td>
</tr>
<tr>
<td>328.fma3d_m</td>
<td>233</td>
<td>192</td>
</tr>
<tr>
<td>330.art_m</td>
<td>79</td>
<td>65</td>
</tr>
<tr>
<td>332.ammp_m</td>
<td>76</td>
<td>48</td>
</tr>
</tbody>
</table>

massive slowdown, but an unusual use case
## Loop parallelism (2)

- loops with OpenMP pragmas only

<table>
<thead>
<tr>
<th>Program</th>
<th>#Loops</th>
<th>#Par.</th>
<th>Main cause of failure</th>
<th>#Priv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>312.swim_m</td>
<td>8</td>
<td>7</td>
<td>reduction</td>
<td>7</td>
</tr>
<tr>
<td>314.mgrid_m</td>
<td>12</td>
<td>11</td>
<td>reduction</td>
<td>11</td>
</tr>
<tr>
<td>316.applu_m</td>
<td>30</td>
<td>17</td>
<td>priv. + reduction</td>
<td>25</td>
</tr>
<tr>
<td>318.galgel_m</td>
<td>37</td>
<td>30</td>
<td>priv. required</td>
<td>30</td>
</tr>
<tr>
<td>320.equake_m</td>
<td>11</td>
<td>3</td>
<td>priv. required</td>
<td>10</td>
</tr>
<tr>
<td>324.apsi_m</td>
<td>28</td>
<td>13</td>
<td>priv. + reduction</td>
<td>27</td>
</tr>
<tr>
<td>326.gafort_m</td>
<td>9</td>
<td>7</td>
<td>priv. + reduction</td>
<td>7</td>
</tr>
<tr>
<td>328.fma3d_m</td>
<td>29</td>
<td>22</td>
<td>reduction</td>
<td>22</td>
</tr>
<tr>
<td>330.art_m</td>
<td>5</td>
<td>4</td>
<td>(non-openmp code)</td>
<td>4</td>
</tr>
<tr>
<td>332.ammp_m</td>
<td>7</td>
<td>5</td>
<td>priv. required</td>
<td>7</td>
</tr>
</tbody>
</table>

- #Priv.: WARs ignored (accesses are collected for feedback)
- very good coverage
- recognizing reductions is hard in the general case
Allen & Kennedy’s *codegen* algorithm
can distribute and re-order loops

```c
void ak(int * X, int * Y, int ** A, int * B, int ** C)
{
    for ( int i=1 ; i<=100 ; i++ ) {
        S1: X[i] = Y[i] + 10;
        for ( int j=1 ; j<=100 ; j++ ) {
            S2: B[j] = A[j][N];
            for ( int k=1 ; k<=100 ; k++ )
                S3: A[j+1][k] = B[j] + C[j][k];
        }
        S4: Y[i+j] = A[j+1][N];
    }
}
```

needs a dependence graph between statements
with dependence levels
Applications > Vectorization (2)

- Target one specific loop
- Keeps dependence type + level
  
  $d$   
  iter. — loop — iter. — $x_1$

  loop — iter. — loop

  iter. — $x_2$

- Resulting dependence graph:

- Combines memory data-dependencies and register traffic
Applications ➤ Linked data structures

- Typically: are the links modified during the traversal of a list?
- Motivation: inspector/executor, speculative parallelization...
- Idea:
  - select a region of interest (e.g., a loop)
  - select memory loads that read an address
    (can be done conservatively by static slicing)
  - capture all RAW dependencies involving one of these loads
- Yesterday’s “Control-Flow Decoupling” is based on such a property

+ Bags of tasks (paper), dependence polyhedra for locality optimizations, ...
Memory (+ control flow) tracing is expensive
  - instrumentation causes code bloat
  - large volume of data

Impacts both tracing and profiling

Sampling does not apply (well)
  - sample memory accesses
  - miss dependencies
  - produces wrong dependencies

Use static analysis
Optimization  >  Static analysis of binary code (1)

- Goal: reconstruct address computations
- Static single assignment form (slicing for free)

```
  mov eax, 0x603140  rax.8 ←
  ...
  sub r13, 0xedb  r13.7 ← r13.6
  ...
  _____  rsi.9 = ϕ(rsi.8, rsi.10)
  ...
  lea r11d, [rsi+0x1]  r11.6 ← rsi.9
  movsxd r10, r11d  r10.9 ← r11.6
  lea rdx, [r10+r13*1]  rdx.15 ← (r10.9,r13.7)
  ...
  lea r9, [rdx+0x... ]  r9.9 ← rdx.15
  ...
  movsd xmm0, [rax+r9*8]  xmm0.6 ← (M.22,rax.8,r9.9)
  ...
```

→ derive symbolic expressions

```
0xe28d4b0 + 8*rsi.9 + ....
```
Scalar evolution (introduces normalized loop counters $I$, ...)

- $0x406ad2$ mov r13.8, qword ptr[...] ; value unknown
- $0x406afd$ r11.93 = phi(...) ; value unknown
- $0x406b05$ mov rdi.97, r11.93 ; = r11.93
- $0x406b10$ rdi.98 = phi(rdi.97, rdi.99) ; = r11.93 + $I \times r13.8$
- $0x406b41$ add rdi.99/.98, r13.8 ; = rdi.98 + r13.8
- $0x406b4a$ j... $0x406b10$

Branch conditions are also parsed (when possible)
- loop trip-counts
Optimization > Memory access coalescing (1)

- Look for accesses to contiguous addresses
  - structure fields
  - unrolling
  - ...
- Inside a basic block only
- Use address expressions
  
  ```
  mov rdx, qword ptr [r13+rdx*8]
  ; → [0x10 + r13_7 + 8*rax_29 - 8*I]
  ...
  mov rax, qword ptr [r13+rax*8]
  ; → [0x8 + r13_7 + 8*rax_29 - 8*I]
  ```
- A single instrumentation point
3 quantities to consider
1. static amount of instrumentation points
2. number of dynamic events
3. run time

SPEC 2006, train (tracing only)
All quantities normalized to the unoptimized case:
Optimization > Parametric loop nests (1)

- Extract static control loops: accesses and control involve
  - loop invariant parameters
  - counters
- Example (436.CactusADM, bench_staggeredleapfrog)

```c
void 0x406b10_1(reg_t r15_58, reg_t r9_81, reg_t r11_93, reg_t rbp_2,
                 reg_t r14_7, reg_t r13_8, reg_t rsi_214, reg_t r10_94)
{
    for ( reg_t I=0 ; (-0x1 + r9_81 + -I >= 0) ; I++ ) {
        if ( (rbp_2 > 0) ) {
            for ( reg_t J=0 ; (-0x1 + rbp_2 + -J >= 0) ; J++ ) {
                ACCESS(’R’, 8, r15_58 + 8*r11_93 + 8*J + 8*r13_8*I);
                ACCESS(’W’, 8, r14_7 + 8*r10_94 + 8*J + 8*rsi_214*I);
            }
        }
    }
}
```

- 8 loop-invariant parameters → instrumented
- no instrumentation on the loop
the loop is compiled and linked to the profiler: 2 cases
  - the loop has an analytical footprint
  - the profiler is responsible for reproducing dependencies
Both optimizations accumulate nicely.

Reduce run time by \( \approx 35\% \).
Conclusion

- A general framework
  - user-selectable dependence domains
  - several decision/parallelization strategies

- A useful tool for (targeted) studies

- Tracing optimizations
  - rely on static analysis
  - applicable to any tracing task