DOE PROXY APPS: COMPILER PERFORMANCE ANALYSIS AND OPTIMISTIC ANNOTATION EXPLORATION

BRIAN HOMERDING
ALCF
Argonne National Laboratory
ECP Proxy Apps

JOHANNES DOERFERT
ALCF
Argonne National Laboratory

April 9th, 2019
Brussels, Belgium
OUTLINE

- Context (Proxy Applications)
- HPC Performance Analysis & Compiler Comparison
- Modelling Math Function Memory Access
- Information and the Compiler
- Optimistic Annotations
- Optimistic Suggestions
ECP PROXY APPLICATION PROJECT

Co-Design

- Improve the quality of proxies
- Maximize the benefit received from their use

Proxy Applications are used by Application Teams, Co-Design Centers, Software Technology Projects and Vendors
PROXY APPLICATIONS

- Proxy applications are models for one or more features of a parent application

- Can model different parts
  - Performance critical algorithm
  - Communication patterns
  - Programming models

- Come in different sizes
  - Kernels
  - Skeleton apps
  - Mini apps

https://proxyapps.exascaleproject.org
The online collection for exascale applications

A major goal of the Exascale Proxy Applications Project is to improve the quality of proxies created by ECP and maximize the benefit received from their use. To accomplish this goal, an ECP proxy app suite composed of proxies developed by ECP projects that represent the most important features (especially performance) of exascale applications will be created.
WHY LOOK AT PROXY APPS

- Proxy applications aim to hit a balance of complexity and usability
- Represent the performance critical sections of HPC code
- Often have various versions (MPI, OpenMP, CUDA, OpenCL, Kokkos)

Issues
- They are designed to be experimented with, they are not benchmarks until the problem size is set
- No common test runner
HPC PERFORMANCE ANALYSIS & COMPILER COMPARISON
PERFORMANCE ANALYSIS
Quantifying Hardware Performance

- Understand representative problem sizes
  - How to scale the problem to Exascale?
- What are the hardware characteristics of different classes of codes? (PIC, MD, CFD)
- Why is the compiler unable to optimize the code? Can we enable it to?
COMPILER FOCUS METHODOLOGY

- Get a performant version built with each compiler
- Identify room for improvement
- Collecting a wide array of hardware performance counters
- Utilize these hardware counters alongside specific code segments to identify areas where we are underperforming
RESULTS
for( int i = 0; i < input.numL; i++ )
{
    phi = data.pseudo_KORS[nuc][i] * sqrt(E);

    if( i == 1 )
        phi -= -atan( phi );
    else if( i == 2 )
        phi -= atan( 3.0 * phi / (3.0 - phi*phi) );
    else if( i == 3 )
        phi -= atan(phi*(15.0-phi*phi)/(15.0-6.0*phi*phi));

    phi *= 2.0;

    sigTfactors[i] = cos(phi) - sin(phi) * _Complex_I;
}
GENERATED ASSEMBLY

Clang

callq  cos
vmovsd %xmm0, 8(%rsp)  # 8-byte Spill
vmovsd 56(%rsp), %xmm0  # 8-byte Reload
    # xmm0 = mem[0],zero
callq  sin
vmovsd .LCPI2_4(%rip), %xmm1  # xmm1 = mem[0],zero
vmovapd %xmm1, %xmm2

GCC

addq  $1, %rbx
addq  $16, %rbp
call  sincos
vpxord %zmm1, %zmm1, %zmm1
vmovsd 40(%rsp), %xmm0
MODELING MATH FUNCTION
MEMORY ACCESS
DESIGN

- Handle the special case
- Model the memory access of the math functions
- Expand Support in the backend
- Expose the functionality to the developer
DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
- Expand Support in the backend
- Expose the functionality to the developer
DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly
- Expand Support in the backend

- Expose the functionality to the developer
DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls

- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly

- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA

- Expose the functionality to the developer
DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls

- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly

- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA
  - Gain coverage of the attribute – Infer the attribute in FunctionAttrs

- Expose the functionality to the developer
DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls

- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly

- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA
  - Gain coverage of the attribute – Infer the attribute in FunctionAttrs

- Expose the functionality to the developer
  - Create an attribute in clang FE
INFORMATION AND THE COMPILER
QUESTIONS

- What information can we encode that we can’t infer?
- Does this information improve performance?
- If not, is it because the information is not useful or not used?
- How do I know what information I should add?
- How much performance is lost by information that is correct but that compiler cannot prove?
 EXAMPLE

>> clang -O3

    int  *globalPtr;
    void external(int*, std::pair<int>&);

    int bar(uint8_t LB, uint8_t UB) {
        int sum = 0;
        std::pair<int> locP = {5, 11};
        external(&sum, locP);

        for (uint8_t u = LB; u != UB; u++)
            sum += *globalPtr + locP.first;

        return sum;
    }

EXAMPLE

>> clang -O3

```c
int *globalPtr;
void external(int*, std::pair<int>&) __attribute__((pure));

int bar(uint8_t LB, uint8_t UB) {
    int sum = 0;
    std::pair<int> locP = {5, 11};
    external(&sum, locP);
    __builtin_assume(LB <= UB);
    for (uint8_t u = LB; u != UB; u++)
        sum += *globalPtr + locP.first; return sum;
}
```
EXAMPLE

```cpp
int *globalPtr;
void external(int*, std::pair<int>&);

int bar(uint8_t LB, uint8_t UB) {
    int sum = 0;
    std::pair<int> locP = {5, 11};
    external(&sum, locP);

    return (UB - LB) * (*globalPtr + 5);
}
```
void baz(int *A);

>> clang -O3 ...

>> verify.sh --> Success
IN A NUTSHELL

```c
void baz(__attribute__((readnone)) int *A);
```

>> clang -O3 ...

>> verify.sh --> Failure
IN A NUTSHELL

```c
void baz(__attribute__((readonly)) int *A);
```

>> clang -O3 ...

>> verify.sh --> Success
OPTIMISTIC OPPORTUNITIES

All Optimistic Opportunities ($\approx 100 - 1000$)

- no-alias on int* A
- readnone/readonly/writeonly on int* C
MARK THEM ALL OPTIMISTIC

All Optimistic Opportunities (≈ 100 - 1000)

1 1 1 1 1 1 3 3 3 3 13 13 13

no-alias on int* A
readnone/readonly/writeonly on int* C
SEARCH FOR VALID

All Optimistic Opportunities ($\approx 100 - 1000$)

- no-alias on int* A
- readnone/readonly/writeonly on int* C
All Optimistic Opportunities ($\approx 100 - 1000$)

1 1 1 1 1 1 3 3 3 3 12 0 0

- no-alias on int* A
- readonly/writeonly on int* C
**OPTIMISTIC CHOICES**

All Optimistic Opportunities ($\approx 100 - 1000$)

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

- no-alias on int* A
- readonly/readonly/writeonly on int* C
OPPORTUNITY EXAMPLE – FUNCTION SIDE-EFFECTS

13. speculatable (and readnone)
12. readnone
11. readonly and inaccessiblememonly
10. readonly and argmemonly
  9. readonly and inaccessiblemem_or_argmemonly
  8. readonly
  7. writeonly and inaccessiblememonly
  6. writeonly and argmemonly
  5. writeonly and inaccessiblemem_or_argmemonly
  4. writeonly
  3. inaccessiblememonly
  2. argmemonly
  1. inaccessiblemem_or_argmemonly
  0. no annotation, original code
ANNOTATION OPPORTUNITIES

- Potentially aliasing pointers
- Potentially escaping pointers
- Potentially overflowing computations
- Potential runtime exceptions in functions
- Potentially parallel loops
- Externally visible functions
- Potentially non-dereferenceable pointers

- Unknown pointer alignment
- Unknown control flow choices
- Potentially invariant memory locations
- Unknown function return values
- Unknown pointer usage
- Potential undefined behavior in functions
- Unknown function side-effects
## OPTIMISTIC TUNER RESULTS

<table>
<thead>
<tr>
<th>Proxy Application</th>
<th>Problem Size / Run Configuration</th>
<th># Successful Compilations</th>
<th># New Versions</th>
<th>Optimistic Opportunities Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSBench</td>
<td>-p 300000</td>
<td>32</td>
<td>9 (28.1%)</td>
<td>225/240 (93.8%)</td>
</tr>
<tr>
<td>XSBench</td>
<td>-p 500000</td>
<td>47</td>
<td>5 (10.6%)</td>
<td>129/141 (91.5%)</td>
</tr>
<tr>
<td>PathFinder</td>
<td>-x 4kx750.adj_list</td>
<td>62</td>
<td>22 (35.5%)</td>
<td>264/299 (88.3%)</td>
</tr>
<tr>
<td>CoMD</td>
<td>-x 40 –y 40 –z 40</td>
<td>49</td>
<td>13 (26.5%)</td>
<td>179/194 (92.3%)</td>
</tr>
<tr>
<td>Pennant</td>
<td>leblancbig.pnt</td>
<td>69</td>
<td>12 (17.4%)</td>
<td>610/689 (88.5%)</td>
</tr>
<tr>
<td>MiniGMG</td>
<td>6 2 2 2 1 1 1</td>
<td>16</td>
<td>4 (25.0%)</td>
<td>479/479 (100%)</td>
</tr>
</tbody>
</table>
# COMPARISON TO LTO

## Performance Gap with LTO as Baseline

<table>
<thead>
<tr>
<th>Proxy Application</th>
<th>LTO</th>
<th>thin-LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSBench</td>
<td>2.86%</td>
<td>5.68%</td>
</tr>
<tr>
<td>XSBench</td>
<td>14.03%</td>
<td>41.23%</td>
</tr>
<tr>
<td>PathFinder</td>
<td>3.67%</td>
<td>4.79%</td>
</tr>
<tr>
<td>CoMD</td>
<td>4.75%</td>
<td>4.48%</td>
</tr>
<tr>
<td>Pennant</td>
<td>-1.13%</td>
<td>-1.14%</td>
</tr>
<tr>
<td>MiniGMG</td>
<td>0.73%</td>
<td>0.79%</td>
</tr>
</tbody>
</table>
OPTIMISTIC SUGGESTIONS
OPTIMISTIC OPPORTUNITIES WITH CHOICES MADE

RSBench
PERFORMANCE CRITICAL OPTIMISTIC CHOICES

RSBench

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 3 0 0 0 0 0
SUGGESTION EXAMPLES

xs_kernel.c:6:1: remark: internalize the function, e.g., through 'static' or 'namespace {}'.

double complex fast_nuclear_W(double complex Z) {
  
In file included from xs_kernel.c:1:
rsbench.h:94:16: remark: provide better information on function memory effects, e.g., through '__attribute__((pure))' or '__attribute__((const))'

complex double fast_cexp(double complex z);
FUTURE WORK

- Improvements to the tool (suggestions and search)
- Additional results
- Identify information that causes regressions
- Understand if information was not useful or not used
- Collect statistics on addition information that does/does not change the binary
ACKNOWLEDGEMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.