Building Source-to-Source Tools for High-Performance Computing

June. 16th, 2020

XSEDE ECSS symposium
* Extreme Science and Engineering Discovery Environment (XSEDE)
* Extended Collaborative Support Service (ECSS)

Dr. Chunhua “Leo” Liao
Agenda

- Background
- Inliner
- Outliner
- The Move Tool
- Other work
- Conclusion
Compilers: traditional vs. source-to-source

Source code

Frontend (parsing)

Middle End (analysis & transformation & optimizations)

Backend (code generation)

Machine code

Source code

Frontend (parsing)

Middle End (analysis & transformation & optimizations)

Backend (unparsing)

Source code

Human-Readable

Modifiable

Portable
ROSE: Enabling Compilers-based Tools for Critical Applications

Binary Executable or Source Code

ROSE Front End
- Source Code Parsing
- Comment + Preproc. Directive + Token Extractors
- Binary Executable Processing

Analysis Results
- Static and Dynamic Analysis

ROSE IR (AST)

AST Transformation

Transformed or Annotated Source Code

ROSE Back End
- Source Code Generation
- Binary Executable Generation

Binary Executable

http://ROSECompiler.org/

June 4, 1996
Guiana Space Centre (CSG)
Kourou, French Guiana

H0+36 seconds
Integer overflow in inertial reference system causes loss of $500 million Ariane 5 launcher and payload.

The U.S. power grid has millions of embedded devices responsible for continuous operation.
Tools Built Using ROSE

- **Analyzers**: understanding, correctness, ...
  - Visualization tools
  - Arithmetic Intensity measuring tool
  - NULL pointer analyzer
  - Data race detection tool

- **Translators**: Optimization, modernization, refactoring, patching ...
  - The AST Inliner
  - The AST Outliner
  - OpenMP Lowering for CPUs/GPUs
  - AutoPar
  - Loop Processor
  - Declaration move tool
  - Code patching tool

Automatically-Generated Machine Chart for multithreaded firmware with 110K SLOC

```c
char destination[5]; char *source = "LARGER";
strcpy(destination, source);
```

Potential reboot

```c
strcpy(destination, source, sizeof(destination));
```

Safe

Identifies unsafe functions and creates patches e.g. detect and repair unsafe strcpy
The AST Inliner for C/C++

- Inlining: replacing a function call with the function body of the called function
  - C++: template functions, some with specialization
  - C++11: lambda expressions (anonymous functions)

- Benefits:
  - Traditional: eliminating overhead of function calls, enabling analysis and optimization which otherwise only work on single functions
  - Source-level C++ inlining: facilitating program understanding, enabling optimizations

Inlining Algorithm

1. Eligibility check:
   a) Only allow named function, static member function, non-virtual member function, with known function body

2. Promoting function call expressions:
   a) e.g. a = func1() + b;  à auto temp = func1(); a = temp + b;

3. Copy the body of the function to be inlined
   a) Create local variables for each formal argument, initialized with the actual argument
   b) Replace variable references with actual arguments
   c) Insert a label to indicate the end of the function body
   d) Convert return x to a code block
      1. E.g.: return x;  à {x; goto func_end;}

4. Postprocessing: cleanup the inlined code
   a) Remove unused labels,
   b) Remove goto to immediate next statement
# Inlining C++ template functions with lambda expressions

<table>
<thead>
<tr>
<th>RAJA</th>
<th>Code Using RAJA</th>
<th>After Inlining*</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace RAJA</td>
<td>void foo()</td>
<td>1. void foo ()</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>2. {</td>
</tr>
<tr>
<td>// Template function</td>
<td>const int n=100;</td>
<td>3. const int n=100;</td>
</tr>
<tr>
<td>template &lt; typename EXE_POLICY_T, typename LOOP_BODY &gt;</td>
<td>void forall( int begin, int end, LOOP_BODY loop_body )</td>
<td>4. double *a = new double [100];</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>5. #pragma novector</td>
</tr>
<tr>
<td>for all ( EXE_POLICY_T ( ), begin, end, loop_body );</td>
<td>}</td>
<td>6. for ( int ii = 0; ii &lt; 15; ++ii ) {</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>7. a[ii] = 0.5;</td>
</tr>
<tr>
<td>// A sequential execution policy type</td>
<td></td>
<td>8. }</td>
</tr>
<tr>
<td>struct seq_exec {}</td>
<td></td>
<td>9. }</td>
</tr>
<tr>
<td>// Template specialization for sequential execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template &lt; typename LOOP_BODY &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void forall( seq_exec, int begin, int end, LOOP_BODY loop_body )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>#pragma novector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for ( int ii = begin; ii &lt; end; ++ ii ) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop_body ( ii );</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The AST Outliner: Effective source-to-source Outlining

Outlining: semantically the reverse transformation of inlining
Used for kernel generation, OpenMP lowering for CPUs and GPUs, autotuning of whole programs

```
#include <omp.h>
#include <stdio.h>

int num_steps = 100000;

int main() {
    double x = 0;
    double sum = 0.0;
    double pi;
    int 1;
    double step = 1.0/(double) num_steps;

    // Run the code in parallel
    #pragma omp parallel for private(i,x) reduction(+:sum) schedule(static)
    for (i=0; i<num_steps; i=i+1) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }

    pi = step*sum;
    printf("%f", pi);
}
```

```
... // omitted headers and a data structure declaration storing variable addresses
static void __out_1__2189__void __out_argv);
int main(int argc, char **argv) {
    ...
}
```

(a) OpenMP program to calculate PI (b) Transformed (or Lowered) code
The AST Outliner: Algorithm and User Interface

- Collect code segments (outlining targets) via interface
- Perform side-effect and liveness analysis
- Bottom up traverse the AST and process each target
  - Check the eligibility of a target
  - Create an outlined function
    - Create a function skeleton with parameters
    - Handle function parameters: decide pass by value vs. reference
  - Move the target into the outlined function’s body
  - Replace variable references: variable cloning to avoid pointer uses
  - Replace the target with a call to the outlined function

Usage: outline [OPTION]... FILENAME...
Main operation mode:
-rose:outline:preproc-only
-rose:outline:abstract_handle handle_string
-rose:outline:parameter_wrapper
-rose:outline:structure_wrapper
-rose:outline:enable_classic
-rose:outline:temp_variable
-rose:outline:enable_liveness
-rose:outline:new_file
-rose:outline:output_path
-rose:outline:exclude_headers
-rose:outline:use_dlopen
-rose:outline:enable_debug

```
// outline the for loop located at line 12 of test3.cpp, call it using dlopen
```
Parameter Handling & Reducing Pointer Dereferences

- **Scope and linkage**
  - C: global only
  - C++: global vs. class-scope, C-linkage

- **Parameters: for control and data**
  - Goal: a few parameters as possible
  - Rely on scope, side effect and liveness analysis

- **Variables pass-by-reference handled by classic algorithms: pointer dereferences**

- **We use a novel method: variable cloning**
  - Check if such a variable is used by address: address-taken analysis
    - C: &x;
    - C++: T & y=x; or foo(x) when foo(T&)
  - Use a clone variable if x is NOT used by address and is assignable

\[
\text{Parameters} = ((\text{AllVars} - \text{InnerVars} - \text{GlobalVars} - \text{NamespaceVars} - \text{ClassVars}) \cap (\text{LiveInVars} U \text{LiveOutVars})) U \text{ClassPointers}
\]

\[
\text{PassByRefParameters} = \text{Parameters} \cap ((\text{ModifiedVars} \cap \text{LiveOutVars}) U \text{ArrayVars} U \text{ClassVars})
\]

\[
\text{CloneCandidates} = \text{PassByRefParameters} \cap \text{PointerDereferencedVars}
\]

\[
\text{CloneVars} = (\text{CloneCandidates} - \text{UseByAddressVars}) \cap \text{AssignableVars}
\]

\[
\text{CloneVarsToInit} = \text{CloneVars} \cap \text{LiveInVars}
\]

\[
\text{CloneVarsToSave} = \text{CloneVars} \cap \text{LiveOutVars}
\]
<table>
<thead>
<tr>
<th>Classic algorithm with pointer-dereferencing</th>
<th>Outlining with variable cloning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void OUT__1__4027__(int *ip__, int *jp__, double omega, double *errorp__, double *residp__, double ax, double ay, double b)</td>
<td></td>
</tr>
<tr>
<td>{ // Four variables becomes pointers: i,j, resid, error</td>
<td></td>
</tr>
<tr>
<td>for (*ip__=1;*ip__&lt;(n-1);(*ip__)++)</td>
<td></td>
</tr>
<tr>
<td>for (*jp__=1;*jp__&lt;(m-1);(*jp__)++)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>*residp__ = (ax * (uold[*ip__-1][*jp__]) + uold[*ip__+1][*jp__]) +</td>
<td></td>
</tr>
<tr>
<td>ay * (uold[*ip__][*jp__-1] + uold[*ip__][*jp__+1]) +</td>
<td></td>
</tr>
<tr>
<td>b * uold[*ip__][*jp__] - f[*ip__][*jp__])/b;</td>
<td></td>
</tr>
<tr>
<td>u[*ip__][*jp__] = uold[*ip__][*jp__] - omega * (*residp__);</td>
<td></td>
</tr>
<tr>
<td>*errorp__ = *errorp__ + (*residp__) * (*residp__);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void OUT__1__5058__(double omega, double *errorp__, double ax, double ay, double b)</td>
<td></td>
</tr>
<tr>
<td>{ int i, j; /* neither live-in nor live-out*/ double resid; /* neither live-in nor live-out <em>/ double error; /</em> clone for a live-in and live-out parameter <em>/ error = <em>errorp__; /</em> Initialize the clone</em>/</td>
<td></td>
</tr>
<tr>
<td>for (i = 1; i &lt; (n - 1); i++)</td>
<td></td>
</tr>
<tr>
<td>for (j = 1; j &lt; (m - 1); j++) {</td>
<td></td>
</tr>
<tr>
<td>resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +</td>
<td></td>
</tr>
<tr>
<td>ay * (uold[i][j - 1] + uold[i][j + 1]) +</td>
<td></td>
</tr>
<tr>
<td>b * uold[i][j] - f[i][j] / b;</td>
<td></td>
</tr>
<tr>
<td>u[i][j] = uold[i][j] - omega * resid;</td>
<td></td>
</tr>
<tr>
<td>error = error + resid * resid;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><em>errorp__ = error; /</em> Save value of the clone*/</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
The Move Tool: a Code Refactoring Tool to Move Variable Declarations into Innermost Scopes

- A source-to-source refactoring tool to support ASC application teams
  - Copy-move variable declarations into innermost scopes: variable privatization
  - Benefits: facilitate code parallelization (migrating to OpenMP/RAJA)

- Algorithm went through 3 versions
  - V1: Naïve single-round move
  - V2: Iterative move using a declaration worklist
  - V3: Separated analysis and movement: much more efficient
Case 1: Single Used Scope vs. Case 2: Multiple Used Scopes

void foo() {
    int i;
    {
        i = 0;
    }
}

DS: Declaration Scope
US: Used Scope
IS: Intermediate Scope

Code with a declaration
a scope tree: three types of Scope Nodes parent-child edges

void foo() {
    int i;
    {
        i = 10;
        {
            i = 0;
        }
    }
}

void foo() {
    int i;
    {
        i = 0;
    }
}

void foo() {
    int i;
    i = 10;
    {
        i = 0;
    }
}

scope tree with multiple used scopes
* trim shadowed used scope
Case 3: Multiple Used Scope Branches of the Same Length

Baseline algorithm V1: handles case 1,2 and 3
Case 4: Multiple Branches with Different Lengths

Algorithm V2: iteratively move declarations

- A declaration copy moved to a new location
  - the newly inserted declaration should be considered for further movements
  - Focus on declarations

- An iterative algorithm using a worklist
  - initial worklist = original declarations in the function
  - while (!worklist.empty())
    - decl = worklist.front(); worklist.pop();
    - moveDeclarationToInnermostScope(decl, inserted_decls);
    - worklist.push_back( each of inserted_decls)

```c
int tmp;
if (tone)
{
    tmp = 0;
}
else
{
    {
        tmp = 0;
    }
}
```

```c
int tmp;
if (tone)
{
    tmp = 0;
}
else
{
    {
        tmp = 0;
    }
}
tmp = g(i);
```

Need further moves
Only Need to Find Final Scopes and Move Once: Algorithm V3

- Find final scopes first
  - `scope_tree_worklist.push(scope_tree);`
  - `while (!scope_tree_worklist.empty())`
    - `current_scope_tree = scope_tree_worklist.front(); ...`
    - `collectCandidateTargetScopes(decl, current_scope_tree);`
      - if (is a bottom scope?)
        - `target_scopes.push_back(candidate)`
      - else
        - `scope_tree_worklist.push_back(candidate)`

- Then copy&move in one shot
  - `if (target_scopes.size()>0)`
    - `copyMoveVariableDeclaration(decl, target_scopes);`
Results

- 230+ regression tests, with correctness verification (diff-based)
- Applied to large-scale X,Y apps, very positive user feedback
- Users kept requesting more features once previous requests were met
  - merge moved declarations with immediately followed assignments
  - transformation tracking, debugging support
  - aggressive mode, keep-going mode, no-op mode, ...
Other work: Benchmarking

If You Can’t Measure it Correctly, You Can’t Improve it

Regression positive/negative tests

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>Confidence of true positive ( P = TP/(TP + FP) )</td>
</tr>
<tr>
<td>Recall</td>
<td>Completeness of true positive ( R = TP/(TP + FN) )</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Chance of having a correct report ( A = (TP + TN)/(TP + FP + TN + FN) )</td>
</tr>
</tbody>
</table>

https://github.com/LLNL/dataracebench


<table>
<thead>
<tr>
<th>Microbenchmark Program</th>
<th>Data Race Detection Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile-time seg. fault (CSF), Unsupported feature (CUN)</td>
<td>Runtime seg. Fault (RSF),</td>
</tr>
<tr>
<td>Runtime timeout (RTO)</td>
<td></td>
</tr>
</tbody>
</table>

Program Analysis and Optimization as Services

Motivation
Hard to use individual tools
Individual tools have limitations

Solution
Compose tools as cloud-based services
Define APIs
Define JSON formats
Automatic Online Training: FreeCompilerCamp.org

Problem:
- Many tools requested by app teams
- But only limited FTEs available

Solution: automatic training and certifying developers
- Modern Learning Management Systems (LMS) + Adaptive Learning/Assessment
- FreeCodeComp → FreeCompilerComp
- Interactive cloud-based playground for learners: Play-with-Docker
Takeaway Messages

ROSE: A source-to-source compiler framework for building tools for national security applications
— tools for source code and binary: inliner, outliner, autopar, move tool, loop processor ...
— http://roseCompiler.org/

Tool Development: right metrics to communicate incremental progress with users
— Regression tests to reflect what users want (positive tests) and don’t want (negative tests)
— Standard metrics to communicate incremental progress with sponsors and users
  • Precision, Recall, Accuracy
— Commenting on issues of apps == commenting on issues of children in front of their parents!

Supportive work
— Benchmarks: people love and hate benchmarks
  • Best qualified people may not want to develop/release the best benchmarks for their work
— Microservice design, docker, cloud,
— Online learning/certifying frameworks, ....