Optimization of X10 Programs with ROSE Compiler Infrastructure

Michihiro Horie†, Mikio Takeuchi†, Kiyokuni Kawachiya†, David Grove‡

†IBM Research - Tokyo ‡IBM T.J Watson Research Center

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923.
X10 compiler

- Front-end executes type checking and AST optimizations
- Back-end generates either C++ or Java source programs
  - Vendor compilers are available for compiling C++ or Java programs

![X10 Compiler Diagram]

**Key Terms**
- AST: Abstract Syntax Tree
- XRX: X10 Runtime in X10
- XRJ: X10 Runtime in Java
- XRC: X10 Runtime in C++
- X10RT: X10 Comm. Runtime

**Diagram Notes**
- Java Interop Support
- Managed X10
  - Java Back-End
  - Java Code Generation
  - Java Source
  - XRJ
  - Java Compiler
  - Existing Java Application
  - Java Bytecode
  - Native Environment (CPU, GPU, etc)
  - X10RT
- Native X10
  - C++ Back-End
  - C++ Code Generation
  - C++ Source
  - CUDA Source
  - XRC
  - Platform Compilers
  - Native executable
  - Existing Native (C/C++/etc) Application
Our goal

- To apply standard optimizations at the front-end and get better performance

- We want to use another compiler that already has the rich set of optimization tools
  - Implementing optimization tools from scratch is a high cost
ROSE compiler infrastructure

- Source-to-source translator
  - Developed in the LLNL
  - Support C/C++/Fortran/Java, MPI/OpenMP, etc.
    - High-level IR optimizations
      - The same level of abstraction as source programs
    - AST consists of both language-common and language-specific IRs
    - Currently there is no mechanism to unify language-specific IRs
      - ROSE unparses back to the same language as the original input language
How to use ROSE’s source-to-source translation

- Users can invoke basic functions such as `frontend()`, `backend()`, etc.
- Also, users need to choose the ordering of ROSE optimizations by invoking their APIs
- Process inherited/synthesized attributes in AST

```c
int main (int argc, char** argv )
{
    SgProject* sageProject = frontend(…);  // starts parsing input source files

    PRE::partialRedundancyElimination(sageProject);  // applies PRE

    ConstantFolding::constantFoldingOptimization(sageProject);  // applies constant folding

    SgFunctionCallExp* functionCall = …
    bool isSucess = doInline(functionCall, true);  // executes inlining for the target function

    generateDOT(*sageProject);  // generates dot file to check generated AST

    return backend(sageProject);  // unparses to source files from the generated AST
}
```
Combining X10 compiler and ROSE

- To enrich optimizations in X10 front-end, we reuse the ROSE optimizations
  - To create a ROSE AST, we convert once-created X10 AST by traversing it
  - Extended ROSE front-end, mid-end, and back-end
    - Reused existing ROSE AST nodes as many as possible to represent X10 code
    - Applied small changes to ROSE optimizations
    - Introduced X10 unparser based on Java unparser

![Diagram of X10 compiler and ROSE integration](image-url)
Front-end

- Defined new ROSE AST nodes for representing APGAS constructs
  - Finish, async, at, etc.
  - Reused the AST nodes common with Java
  - Reused the AST nodes common with C++ for closure, struct, etc.

- Valid AST conversion was necessary to pass ROSE semantic analyses
  - Constructed type hierarchies also in ROSE

- Only the library classes that are directly referenced in input classes are parsed
Preparation for the mid-end

- Type conversion was necessary to use ROSE optimizations
  - X10 uses object types to represent arithmetic types, while ROSE uses C++ types inside each corresponding ROSE AST nodes
    - \texttt{x10.lang.Long} \rightarrow \texttt{long}
    - \texttt{x10.lang.Rail[Long]} \rightarrow \texttt{long[]}
  - They are all final classes

- Also replaced the method invocations of converted types to the one of static helper functions, which has no method body
  - \texttt{Rail.size()} \rightarrow \texttt{X10\_ROSE\_Helper\_Rail.size()}

- In the unparser, the converted types and helper functions changed back to X10 data types
Mid-end

- Loop optimizations worked as-is
  - ROSE’s mid-end mainly supports C or C++
  - AST representation for loop blocks does not change among X10 and C/C++

- Some optimizations were needed to change its strategy to align with X10
  - ROSE’s default inliner uses *goto* statements for return-statements. Instead, we changed not to use goto.

```scala
import java.util.*

class InliningExample {
  def cond(a: Long): Boolean = a < 8
  def main(args: Array[String]) {
    val o = new InliningExample()
    var i: Long = 0
    for (; o.cond(i); ++i) {
      // loop body
    }
  }
}
```
An example of using the mid-end API

```cpp
SgJavaClassDeclarationList *class_list = file->get_class_list();
vector<SgClassDeclaration *> &type_list = class_list->get_java_class_list();
for (int i = 0; i < type_list.size(); i++) {
    SgClassDeclaration *class_declaration = type_list[i];
    SgClassDefinition *class_definition = class_declaration->get_definition();
    AstSgNodeListAttribute *attribute = (AstSgNodeListAttribute *) class_definition->getAttribute("class_members");
    for (int j = 0; j < attribute->size(); ++j) {
        SgFunctionDefinition *method_definition = isSgFunctionDefinition(attribute->getNode(j));
        SgBasicBlock *stmts = method_definition->get_body();
        SgStatementPtrList &stmts2 = stmts->get_statements();
        for (SgStatementPtrList::iterator m = stmts2.begin(); m != stmts2.end(); ++m) {
            if (isSgBasicBlock(*m)) {
                SgBasicBlock *bb = (SgBasicBlock *)m;
                SgStatementPtrList &stmts3 = bb->get_statements();
                for (SgStatementPtrList::iterator l = stmts3.begin(); l != stmts3.end(); ++l) {
                    if (isSgForStatement(*l)) {
                        SageInterface::loopUnrolling((SgForStatement *)l, (size_t)4);
                    }
                }
            }
        }
    }
}
```
Available as OSS

- ROSE side support on Github
  - [https://github.com/rose-compiler/edg4x-rose/tree/master/src/frontend/X10_ROSE_Connection](https://github.com/rose-compiler/edg4x-rose/tree/master/src/frontend/X10_ROSE_Connection)

- X10 side support in X10 sourceforge
  - [http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.compiler/src/x10rose](http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.compiler/src/x10rose)

- ROSE installation became much easier by using our installation scripts on the Github
  - A week for trial and error ➔ a few hours, just waiting!

- We confirmed that our system works on
  - CentOS 6.5
  - Red Hat Enterprise Linux 6.1
  - Red Hat Enterprise Linux 5.11
Benchmarks

- **X10 tests in** [http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.tests/tests/](http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.tests/tests/)
  - More than 2,000 files
  - To see the coverage of current parser and unparsers

- **Proxy applications**
  - HPC applications that are developed in the research projects of U.S. Department of Energy (DoE)
  - Implemented in C/C++, Fortran with MPI, OpenMP, etc.
    - LULESH: Lagrangian hydrodynamics
    - MCCK: neutronics, investigating the communication cost
    - CoMD: molecular dynamics
  - We have X10 port of these proxy applications
    - See [http://sourceforge.net/p/x10/code/HEAD/tree/applications/trunk](http://sourceforge.net/p/x10/code/HEAD/tree/applications/trunk)
    - Also we have a publication:
      
Coverage on parser and unparserr (without optimizations)

- Currently 73% on average
  - Success: unparsed X10 code was able to compile without an error
Applying ROSE optimizations to LULESH

- X10 port of LULESH revision 28354

- Currently, we tried to apply basic optimizations
  - Loop unrolling
  - Method inlining
  - Stack allocation
  - LoopInvariant Hoisting

- Estimated how much execution performance can improve by using ROSE optimizations
Loop unrolling

- ROSE provides an API:
  - `loopUnrolling(SgForStatement *loop, size_t factor)`
- We found 6 targets in LULESH

```java
protected def calcForceForNodes(domain : Domain) {
  var i_nom_5 : long;
  val _lu_fringe_6 = domain.numNode - 1 -
  ((domain.numNode == 0 ? domain.numNode :
    (domain.numNode + 1)) % 4 == 0 ? 0 : 4);
  for (i_nom_5 = 0L; i_nom_5 <= _lu_fringe_6; i_nom_5 += 4) {
    domain.fx(i_nom_5) = 0.0;
    domain.fy(i_nom_5) = 0.0;
    domain.fz(i_nom_5) = 0.0;
    domain.fx(i_nom_5 + 1) = 0.0;
    domain.fy(i_nom_5 + 1) = 0.0;
    domain.fz(i_nom_5 + 1) = 0.0;
    domain.fx(i_nom_5 + 2) = 0.0;
    domain.fy(i_nom_5 + 2) = 0.0;
    domain.fz(i_nom_5 + 2) = 0.0;
    domain.fx(i_nom_5 + 3) = 0.0;
    domain.fy(i_nom_5 + 3) = 0.0;
    domain.fz(i_nom_5 + 3) = 0.0;
  }
  for (; i_nom_5 <= (domain.numNode - 1L); i_nom_5 += 1) {
    domain.fx(i_nom_5) = 0.0;
    domain.fy(i_nom_5) = 0.0;
    domain.fz(i_nom_5) = 0.0;
  }
}
```
Inlining (@Inline)

- ROSE provides an API:
  - `doInline(SgFunctionCallExp* funcall, bool allowRecursion)`

- We found 16 targets in LULESH

```python
def calcVolumeForceForElems(domain : Domain) {
    val hgcoef = domain.hgcoef;
    val determ = new Rail[double](numElem);
    :
    calcHourglassControlForElems(domain,determ,hgcoef);
}

def calcHourglassControlForElems(domain : Domain, determ : Rail[double], hgcoef : double) {
    val numElem = domain.numElem;
    val numElem8 = numElem * 8L;
    if (dvdx == null) {
        dvdx = new Rail[double](numElem8);
        dvdy = new Rail[double](numElem8);
        dvdz = new Rail[double](numElem8);
        :
    }
    :
}

Before optimization
```
Stack allocation

- ROSE does not provide optimization API for applying stack allocation
- Instead, we would like to attach `@StackAllocate` of X10 automatically
  - We check whether attaching `@StackAllocate` is valid
    - Reusing pointer analysis that ROSE provides
- In LULESH, 26 variables were found to be stack-allocatable
  - These variables were all `Rail` objects, and they were declared within the main loop of calculation

Before optimization
```
val x1 = new Rail[double](8L);
```

After optimization
```
@StackAllocate val x1 = @StackAllocate new Rail[double](8L);
```
Loop invariant hoisting

- Although ROSE does not provide optimization API, we can realize by using primitive ROSE APIs
  - `insertStatement()`
  - `removeStatement()`

- We found 1 target in LULESH

- To decide whether hoisting loop invariants is valid or not, we can use ROSE’s analyses:
  - Use-Definition analysis
  - Liveness analysis

```java
val numElem = domain.numElem;
val numElem8 = numElem * 8L;
while ((domain.time < domain.stopTime) &&
       (domain.cycle < Lulesh.this.opts.its)) {
    dvdx = new Rail[double](numElem8);
    dvdy = new Rail[double](numElem8);
    dvdz = new Rail[double](numElem8);
}
```

Before optimization

```java
val numElem = domain.numElem;
val numElem8 = numElem * 8L;
dvdx = new Rail[double](numElem8);
dvdy = new Rail[double](numElem8);
dvdz = new Rail[double](numElem8);
while ((domain.time < domain.stopTime) &&
       (domain.cycle < Lulesh.this.opts.its)) {
    :
}
```

After optimization
Execution performance in a single place and single process

- By incrementally applying optimizations, we observed a 10% performance improvement
  - This is also a 2% improvement compared to the C++ original

<table>
<thead>
<tr>
<th></th>
<th>LULESH elapsed time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>X10-port</td>
</tr>
<tr>
<td>(2)</td>
<td>(1) + inlining</td>
</tr>
<tr>
<td>(3)</td>
<td>(2) + loop unrolling</td>
</tr>
<tr>
<td>(4)</td>
<td>(3) + stack allocation</td>
</tr>
<tr>
<td>(5)</td>
<td>(4) + loop invariant hoisting</td>
</tr>
</tbody>
</table>

P7IH: (32 Power7 cores 3.84 GHz, 128 GB memory, peak: 982 Gflops) * 1 nodes
Red Hat Enterprise Linux Server release 6.4 (Santiago)
X10 2.5.1 native backend (compile option: -X10rt pami –O –NO_CHECKS)
C/C++ compiler : XL C V12.1 (compile option: -O3 -qinline)
Conclusion

- Applying standard optimizations at the front-end and getting better performance
  - We want to use a different compiler that already has the rich set of optimization tools
  - To implement optimization tools from scratch is a high cost

- We estimated the execution performance of LULESH in a single place and single process
  - Improved by 10% compared to the X10-port

- Future works
  - We have to extend ROSE because it is not APGAS-oriented
  - Also, we should support APGAS-specific optimizations in ROSE