Leveraging Model-Driven Engineering Techniques in Optimizing Compiler Research

Tomofumi Yuki, Sanjay Rajopadhye, Robert France
Colorado State University

Antoine Floch, Clement Guy, Steven Derrien, Benoit Combemale
University of Rennes 1, IRISA, INRIA
Optimizing Compilers

- Goal: generate “efficient” code
  - Execution time
  - Energy consumption
  - Code size
- Wide range of optimizations
  - Register allocation
  - Dead code elimination
  - Automatic parallelization
  - Run-time optimizations
Optimizing Compiler Research

- Prototype implementations
  - “Proof of Concept”
  - Evaluation

- Compilers are complicated pieces of software
  - Need for rapid development
  - Development spans generations of students
  - Performance of compiler prototype not critical
Compiler Community Tools

- Existing infrastructure focus on C/Fortran
  - SUIF, ROSE, Cetus
  - Extending the language takes time and effort
  - Cannot be used for new languages

- Parser generators
  - Yacc, Bison, ANTLR
  - Generates parser, classes for AST, and visitor
    - but this is usually how far tools get
Outline

- Introduction
- Bridging MDE with Research Compilers
  - Challenges in compilers
  - Corresponding MDE tools
- Stencil Compiler
- Conclusions
Optimizing Compiler Examples

- High-level flow of two research compilers:

  1. Parse source language
  2. Transform intermediate representations (IRs) for efficiency. May take domain specific knowledge as additional inputs.
  3. Output code or binary
Research Compiler Challenges

- Maintainable and Sustainable Code
  - Developers may not have good SE background
- Analysis/Query of IRs
  - Is the IR consistent after parsing/transformation?
  - Find where to apply transformations
- Code Generation
- Use of Domain Specific Knowledge
Bridging with MDE

- View compiler IRs as models

**DSLs and Tooling**

**Model Transformations and Analyses**

**Code Generation**

- Parsing
- DSL IR
- Optimizations
- Analyses
- Execution Strategies
- CodeGen Model
- Code Generation Framework
- C
- C+OpenMP
- C+MPI

- Parsing
- C IR
- Target Independent Optimizations
- Partitioning
- Platform Specification
- Processor Description
- Speed/Area Constraints
- HW Accelerator Generation
- HW Model
- SystemC
- VHDL
- Binary for ASIP
Outline

- Introduction
- Bridging MDE with Research Compilers
- **Stencil Compiler**
  - Domain specific compiler
  - Examples of DSL evolution
- Conclusions
Stencil Compiler

- Highly specialized compiler for stencils
  - Many specific characteristics to leverage
    - Difficult to deduce in general compilers

- Goals:
  - Simple DSL for specifying stencils
  - Generate different variations with minimal effort
  - Provide highly optimized code
Stencil Computations

- A class of programs that encompasses many applications
  - Iteratively updates $d$-dimensional data using neighboring values

\[ \ldots \]
\[ t=2 \]
\[ t=1 \]
\[ t=0 \]
Stencil Variations

- Many variations and optimization choices
  - Stencil pattern, update equations
  - Update scheme: Jacobi, Gauss-Seidel, chaotic
  - Boundaries: no-update, periodic, symmetric
Desired Development Process

- Incremental
  - Start simple, and keep adding new features
    - Can we frequently update the DSL?

- Modular
  - Many different directions for adding features
    - Can we have modularized additions to the DSL?
Role of MDE

- Non-MDE tools are available for generating parsers and code generators
  - May not fit each other well
  - Changing the DSL requires effort
- MDE tools revolve around models
  - Provide consistency across tools
  - Much easier to make incremental changes
Example: 1D Stencil

Meta-Model

```plaintext
data Stencil
  data grids : DataGrid
  data sizeParams : Symbol
  data timeParam : Symbol

data DataGrid
  data name : EString
  data size : Symbol
  data update : UpdateEquation

data Symbol

data UpdateEquation
```

DSL

```plaintext
stencil ( N ) {
  time:T;
  var A {
    size:[N];
  }
}
```

Code

```plaintext
for (t=0; t<T; t++)
  for (i=1; i<N-1; i++)
```
Example: adding time

- Now an update uses values from previous $t$

```plaintext
stencil ( N ) {
    time:T;
    var A {
        size:[N];
    }
}
```

- You need to change memory size and accesses

```plaintext
for (i=1; i<N-1; i++)
    A[t%2,i] = update(i, i+1, i-1);  
    (A[(t-1)%2,i]+A[(t-1)%2,i+1]+A[(t-1)%2,i-1]);
```

Body of update replaced for brevity
Example: adding boundaries

- Now let's add periodic boundaries

```plaintext
stencil ( N ) {
    time:T;
    var A {
        size:[N];
    }
}
```

- Corresponding code becomes more complex

```plaintext
for (t=0; t<T; t++)
    A[t%2,0] = update(0,1,N-1);
for (i=1; i<N-1; i++)
    A[t%2,i] = update(i,i+1,i-1);
A[t%2,N-1] = update(N-1,0,N-2);
```
Example: adding tiling

- Lets do tiling (optimizing transformation)

```plaintext
 stencil ( N ) {
     time: T;
     var A {
         size: [N];
     }
     tiling: [10, 10];
 }
```

- Simple specification, complex code

```plaintext
 for (tt=0; tt<T; tt+=10)
     for (ti=1+tt; ti<tt+N-1; ti+=10)
         for (t=tt; t<min(tt+10, T); t++)
             for (i=ti; i<min(ti+10, t+N-1); i++)
                 A[t%2, i-t] = update(i-t, i-t+1, i-t-1);
```
Code Generator

- Based on Xtend2
- 3 core modules
  - Loop nest
  - Memory allocation and access
  - Update statements
- Injection pattern for maximizing reuse
  - Many options can be “toggled” easily
    - e.g., Linearlized array / multi-dimensional array
Outline

- Introduction
- Bridging MDE with Research Compilers
- Stencil Compiler

**Conclusions**
- Limitations of MDE
- Remaining challenges
Limitations of MDE

- MDE helps us in building compiler prototypes
- MDE does not let you do something “new”
  - It may take more time, but you can do it
- MDE does not solve your problem
  - Cannot make bad idea to work
  - Modeling needs to be done “right”
Some Remaining Challenges

- Model transformation verification and testing
  - Semantic preserving model transformations
- Coping with a multitude of DSLs
  - Avoid defining and tooling of a DSL from scratch
  - Inter-language reasoning
- Scalability
  - Compilers handle large models
Conclusion

- Compiler IRs = models
  - MDE tools for building compilers
  - Appealing for research compilers

- DSL Compilers
  - Our example: stencil computation
  - Area of high interest, but compiler community lack the tools
Challenges

- Analyses and Manipulation of IRs

Structural Validation

Complex Querying and Transformations

- DSL
  - Parsing
  - DSL IR
    - Optimizations
      - Analyses
        - Execution Strategies
          -CodeGen Model
            - Code Generation Framework
              - C
              - C+OpenMP
              - C+MPI

- C
  - Parsing
  - C IR
    - Target Independent Optimizations
      - Partitioning
        - Speed/Area Constraints
          - Platform Specification
            - Processor Description
              - ASIP Code Generation
                - HW Accelerator Generation
                  - HW Model
                    - SystemC
                    - VHDL
                      - Binary for ASIP

Interaction
Challenges and MDE Solutions

Analyses and Manipulation of IRs

- Analyses and Manipulation of IRs
  - Structural Validation
    - Structural Properties on models (conformity)
    - OCL constraints
  - Complex Querying and Transformation
    - OCL queries
    - M2M tools
    - Rewriting rules

Diagram:
- DSL IR
- Optimizations
- Analyses
- Execution Strategies
- CodeGen Model
- Code Generation Framework
- C
- C+OpenMP
- C+MPI
- Parsing
- Target Independent Optimizations
- Partitioning
- Speed/Area Constraints
- Processor Description
- Platform Specification
- ASIP Code Generation
- Binary for ASIP
- HW Model
- VHDL
- SystemC
- HW Accelerator Generation
Challenges

- Domain specific knowledge is heavily utilized
Challenges and MDE Solutions

- Domain specific knowledge is heavily utilized

MDE-based DSLs
Generative approaches (editor, parser)
Challenges

- Code generation and external tools

Use of External Tools (term rewriting, ML, LP, CSP, ...)

Code Generation Framework

- C
- C+OpenMP
- C+MPI
Challenges and MDE Solutions

- Code generation and external tools

- Use of External Tools (term rewriting, ML, LP, CSP, …)

- Metatools
  - Metamodel instrumentation
  - Defining new generative tools