Open Language Implementation

or: How I Learned to Stop Worrying and Love My Compiler

Robert M. Fuhrer
Mandana Vaziri
Evelyn Duesterwald
IBM Research

IWMSE 2011
Outline

- What makes parallel programming hard
- Why current tools don't help enough
- 3 tool characteristics that might help
- Examples
Writing Parallel Programs is Hard (duh)

- Functional correctness
  - Understand what code executes in parallel
  - Understand the memory model (who sees what when)
  - Understand shared variables, invariants, synchronization
  - Manage data/control dependencies
  - Understand potentially huge space of possible interleavings

- Performance
  - Understand data distribution/layout
  - Understand relative cost of operations vs. "orchestration"
Understanding Parallel Execution is Hard (duh)

- Non-determinism due to:
  - Variable timing introduced at any level influences task scheduling
  - Racing accesses to shared variables may cause:
    - real errors (lack of synchronization) or
    - innocuously different results (just confusing)

- Apparent non-determinism (complex, unpredictable, even chaotic tool behavior) due to, e.g.:
  - Compiler pipeline optimizations
  - Hardware-level reordering
Understanding Parallel Compilers is Hard (duh)

- Compiler analyzes code, performs many types of optimizations:
  - Inlining
  - Specialization
  - Reordering
  - Loop restructuring (e.g. polyhedral transformations)
  - Parallelization (sometimes)
- Solution space is colossal
- Optimality of a given solution difficult to predict, even relative to well-characterized inputs
Key Issues w/ Current Tools

Common user’s questions:

‣ “What did the compiler do to my code?”
‣ “Why did/didn’t the compiler do this to my code?”
‣ “How do I get the compiler to apply that optimization?”

High-level languages actually **exacerbate** the problem by increasing the gap betw. source & impl, leaving even more choices!
A Call to Arms: Compensation via Tooling

User-visible “open-ness”:

- Transparency: “what the tool did”
- Accountability: “why the tool did what it did”
- Interactivity: “how to control the tool behavior”

Also: open-ness from tool-builder’s perspective

- Need to extract building blocks that are deeply intertwined in the compiler, refactorings, ..., and make them composable
- Instrument analyses, transformations with user-level smarts to make them suitable for building a wide variety of tooling
Property 1: Transparency ("what")

- The set of optimizations that was applied by the compiler and/or runtime is usually hidden from the developer.

- Resulting program behavior can be markedly different from what was originally specified (even with an imperative language).

- This knowledge is critical to the developer's understanding of the program's current and potential behavior.
  
  - e.g., the set of possible interleavings.

- This knowledge could aid the developer's understanding of functionality and performance bottlenecks.
  
  - Make this info accessible to the developer in a consumable way!
Property 2: Accountability ("why")

- The compiler (and runtime) make many decisions/determinations:
  - The validity of applying a transformation
  - The benefit of applying a transformation
- This information is typically lost
- This information can help the developer understand why their program performs as it does, and what they need to do to fix it
- Make this info accessible to the developer in a consumable way!
Property 3: Interactivity ("how")

- Many compilers provide a set of coarse-grained and global knobs to influence code optimization
  - "-O3"
  - "-finline-functions"
  - "-funroll-loops"
- Optimization space is huge, and heuristics favor certain loads
  - Missed opportunities
  - Optimizations that actually hurt performance
- Give selective control back to the user in a consumable way
Examples

- Transparency in compilation
- Accountability in compilation
- Accountability of compiler semantic checks
- Accountability of concurrency/performance refactorings
- Interactivity for concurrency/performance refactorings
User's perspective of program is the source

Feedback needed at that level

Each transformation retained as a first-class entity with args

If transformations compose in a "declarative" way -> easier

Source-to-source (e.g., AST-level) transformations [easier]:

If you use a rewrite engine, this is probably not too hard

If you write imperative transformation code, it's harder

Transformations on intermediate representations [harder]:

Need to correlate IR back to original source (nontrivial)
Accountability: Compilation

Say why the compiler did X

- Ease depends on the kind of optimization framework
- If individual decisions made incrementally → easier (?)
- All decisions made together (e.g. by a solver) → hard to tease apart (“I don’t know why… it’s just optimal”)

Say why the compiler did not do Y

- Many possible optimizations → not feasible/sensible to say why all of the things that didn’t happen didn’t happen
- But: if compiler permits user to suggest optimizations, this can be done more reasonably, and is highly desirable
- Precondition not met for requested Y? → Say so, w/ supporting evidence (so user can do something about it)
Accountability: Semantic Checks

Type checkers for complex type systems (generics, dependent types) employ constraint solvers (SAT, SMT, iterative fixed-point)

Explaining the solver’s results is critical to language usability!

Constraint solvers use an abstraction of the source structure

Constraint graph paths correspond to program data/control flows

Possibilities:

- Make paths explicit in explanations (some compilers do this)
- Better(?): Allow user to explore the constraint system
- Even cooler: Tie constraints to lang. semantics (teach user!)
- Such information is useful for other tools! (IDE content assist,...)
Accountability: Concurrency/Performance Refactoring

- Refactoring ::= “behavior-preserving src-to-src transformation”
- Refactoring ::= safety preconditions + transformation
- When preconditions fail, current tools typically give very poor (or no) explanations
- Leaves user wondering why he can’t apply the transformation, doesn’t know how to “fix” things
- Many “interesting” refactorings perform nontrivial static analysis to check transformation preconditions (e.g., type inference, data-/control-flow analysis, dependency analysis)
- Analyses not instrumented properly to explain their findings
Refactorings are “smart” b/c they’re safe

Refactorings are “dumb”: they know how to do just one thing

Preconditions may fail b/c the user needs to prepare with an enabling refactoring (say, to make a structural feature evident)

Trivial example: inappropriate visibility in “Move Member”

But most refactorings won’t suggest enabling refactorings

User loses access to analysis results if they try it manually

Preconditions may fail b/c the behavior can’t be preserved

But maybe the “offending” behavior isn’t intended!

Analysis results may suggest a “fix” for unintended behavior
Summary

Language tools are especially complex for parallel software

Need these tools to be more communicative about their results

Need more selective control over their behavior

Two aspects of openness

- Openness from the user’s perspective
  - Tools should be transparent (“what”), accountable (“why”), interactive (“how to control”)

- Openness from the tool-builder’s perspective
  - Need the right kind of building blocks as tool enablers
  - Even low-level building blocks need user-centric design