Compiler Transformations for High-Performance Computing (1)

Presented by
Jason Pazis and Yi Zhang

March 23, 2010
What’s this survey about?

- Comprehensive overview of *high-level* compiler transformations/optimizations
- Languages: imperative, e.g. C, Fortran
- Architectures
  - Sequential: common and general-purpose
  - Parallel: superscalar, vector, SIMD, shared-memory MP, distributed-memory MP, etc
What do compilers do?

- On a high level
  - Translation: source code $\rightarrow$ machine code
  - Optimization: various transformations to reduce running time, code size, etc

Clear separation of high-level programming languages and machine architecture
What do compilers do?

- **On a high level**
  - Translation: source code → machine code
  - Optimization: various transformations to reduce running time, code size, etc

- **Specifically**
  - Lexical analysis
  - Parsing
  - Semantic Analysis
  - Optimization
  - Code generation

Clear separation of high-level programming languages and machine architecture
What do compilers do?

- On a high level
  - Translation: source code → machine code
  - Optimization: various transformations to reduce running time, code size, etc

- Specifically
  - Lexical analysis
  - Parsing
  - Semantic Analysis
  - Optimization
  - Code generation

Clear separation of high-level programming languages and machine architecture
Optimization trilogy

Decide → Verify → Transform
Decide

- Difficult and poorly understood
  - Search space is huge
  - Decision making is complicated and expensive: some are NP-complete or even undecidable

```c
int foo (void) {
  signed char x = 1;
  unsigned char y = -1;
  return x > y;
}
```
Decide

- Difficult and poorly understood
  - Search space is huge
  - Decision making is complicated and expensive: some are NP-complete or even undecidable
- Mostly a collection of piecemeal heuristics
  - With some ordering heuristics
  - With some progress in systematic application of families of transformations
Decide

- Difficult and poorly understood
  - Search space is huge
  - Decision making is complicated and expensive: some are NP-complete or even undecidable
- Mostly a collection of piecemeal heuristics
  - With some ordering heuristics
  - With some progress in systematic application of families of transformations
- Conflicts not uncommon, leading to
  - Worse performance: less code → less efficient use of cache
  - Incorrect program: e.g., Ubuntu 8.04’s patch made the following code always output 1

```c
int foo (void) {
    signed char x = 1;
    unsigned char y=-1;
    return x > y;
}
```
Scope of decision

- Statement
- Basic block (straight-line code)
- Innermost loop
- Perfect loop nest
- General loop nest
- Procedure (aka global optimization)
- Interprocedural
What is a legal transformation? (Given original program A and transformed program B)

- B and A perform exactly the same operations in the same order
What is a legal transformation? (Given original program A and transformed program B)

- B and A perform exactly the same operations in the same order — too strict
What is a legal transformation? (Given original program A and transformed program B)

- B and A perform exactly the same operations in the same order — too strict
- B and A produce exactly the same output for all identical executions
  - With same input data
  - With same results for nondeterministic operations, e.g, \texttt{rand()}
What is a legal transformation? (Given original program A and transformed program B)

- B and A perform exactly the same operations in the same order — too strict
- B and A produce exactly the same output for all identical executions — still too strict
  - With same input data
  - With same results for nondeterministic operations, e.g, rand()
Let’s verify

(a) Original

```latex
do \ i=1, n \\
\hspace{1cm} a[i] = b[k]+a[i]+100000.0 \\
end do \\
return
```

(b) Transformed

```latex
C = b[k]+100000.0 \\
do \ i=n,1,-1 \\
\hspace{1cm} a[i] = a[i]+C \\
end do \\
return
```

Problems:

▶ Evaluating $C$ first may cause overflow
▶ Reordered additions of float-point numbers may cause different results
▶ Algebraic commutative operations can be computationally non-commutative for float-point numbers ((semicommittive))
▶ If $k$ is out of range of array $b$, memory fault can happen at a different place
▶ $a$ and $b$ may be completely or partially aliased to one another, causing updated $b[k]$ to be used in (a) but not in (b)
Let’s verify

(a) Original

do i=1,n
    a[i] = b[k]+a[i]+100000.0
end do
return

(b) Transformed

C = b[k]+100000.0
do i=n,1,-1
    a[i] = a[i]+C
end do
return

Problems:

- Evaluating C first may cause overflow
Let’s verify

(a) Original

do i=1,n
  a[i] = b[k]+a[i]+100000.0
end do
return

(b) Transformed

C = b[k]+100000.0

do i=n,1,-1
  a[i] = a[i]+C
end do
return

Problems:

- Evaluating \( C \) first may cause overflow
- Reordered additions of float-point numbers may cause different results
  - Algebraic commutative operations can be computationally non-commutative for float-point numbers (semicommutative)
- If \( k \) is out of range of array \( b \), memory fault can happen at a different place
- \( a \) and \( b \) may be completely or partially aliased to one another, causing updated \( b[k] \) to be used in (a) but not in (b)
Let’s verify

(a) Original

\[
\begin{align*}
do & \ i=1,n \\
a[i] & = b[k]+a[i]+100000.0 \\
end & \ do \\
return & 
\end{align*}
\]

(b) Transformed

\[
\begin{align*}
C & = b[k]+100000.0 \\
do & \ i=n,1,-1 \\
a[i] & = a[i]+C \\
end & \ do \\
return & 
\end{align*}
\]

Problems:

- Evaluating $C$ first may cause overflow
- Reordered additions of float-point numbers may cause different results
  - Algebraic commutative operations can be computationally non-commutative for float-point numbers (semicommutative)
- If $k$ is out of range of array $b$, memory fault can happen at a different place
Let’s verify

<table>
<thead>
<tr>
<th>(a) Original</th>
<th>(b) Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=1,n</td>
<td>C = b[k]+100000.0</td>
</tr>
<tr>
<td>a[i] = b[k]+a[i]+100000.0</td>
<td>do i=n,1,-1</td>
</tr>
<tr>
<td>end do</td>
<td>a[i] = a[i]+C</td>
</tr>
<tr>
<td>return</td>
<td>end do</td>
</tr>
<tr>
<td></td>
<td>return</td>
</tr>
</tbody>
</table>

Problems:

- Evaluating $C$ first may cause overflow
- Reordered additions of float-point numbers may cause different results
  - Algebraic commutative operations can be computationally non-commutative for float-point numbers (*semicommutative*)
- If $k$ is out of range of array $b$, memory fault can happen at a different place
- $a$ and $b$ may be completely or partially aliased to one another, causing updated $b[k]$ to be used in (a) but not in (b)
So how to ensure correctness in practice?

- Having different levels of “correctness”
  - Original & transformed produce bitwise-identical results for identical executions
  - Original & transformed perform equivalent operations for identical executions
    - All permutations of semicommutative operations are considered equivalent
    - May produce not bitwise-identical results
So how to ensure correctness in practice?

- Having different levels of “correctness”
  - Original & transformed produce bitwise-identical results for identical executions
  - Original & transformed perform equivalent operations for identical executions
    - All permutations of semicommutative operations are considered equivalent
    - May produce not bitwise-identical results
- Enforcing restrictions in the programming language
  - Fortran forbids argument aliases in function calls
Typical goals of transformations

- Maximize use of computational resources
  - May not be true for embedded, resource-constrained devices
- Minimize the number of operations performed (fewer machine cycles)
- Minimize use of memory bandwidth (e.g., fewer cache misses)
- Minimize size of total memory required (both code & data sizes)
Compiler Organization

- Optimization takes place in three distinct phases
  - High-level intermediate language
  - Low-level intermediate language
  - Object code

Where is each one of these levels most useful?

- High-level intermediate language
  - Higher-level transformations
    - Example: Array references vs low-level address calculations

- Low-level intermediate language
  - Low-level machine independent transformations
    - Example: Address computations

- Object code
  - Machine specific optimizations
    - Example: Binary-to-binary translations
Compiler Organization

- Optimization takes place in three distinct phases
  - High-level intermediate language
  - Low-level intermediate language
  - Object code
- Where is each one of these levels most useful?

Example: Array references vs low-level address calculations

Example: Address computations

Example: Binary-to-binary translations
Compiler Organization

- Optimization takes place in three distinct phases
  - High-level intermediate language
  - Low-level intermediate language
  - Object code

- Where is each one of these levels most useful?
- High-level intermediate language
  - Higher-level transformations
  - Example: Array references vs low-level address calculations
Compiler Organization

- Optimization takes place in three distinct phases
  - High-level intermediate language
  - Low-level intermediate language
  - Object code
- Where is each one of these levels most useful?
- High-level intermediate language
  - Higher-level transformations
  - Example: Array references vs low-level address calculations
- Low-level intermediate language
  - Low-level machine independent transformations
  - Example: Address computations $a[5, 3], a[7, 3]$
Compiler Organization

- Optimization takes place in three distinct phases
  - High-level intermediate language
  - Low-level intermediate language
  - Object code

- Where is each one of these levels most useful?
  - High-level intermediate language
    - Higher-level transformations
    - Example: Array references vs low-level address calculations
  - Low-level intermediate language
    - Low-level machine independent transformations
    - Example: Address computations $a[5, 3], a[7, 3]$
  - Object code
    - Machine specific optimizations
    - Example: Binary-to-binary translations
Dependence analysis

- What is a dependence?
  - A relationship between two computations
  - Places constraints on their execution order

- Two kinds of dependences
  - Control dependences
    - if (a == 3)
    - b = u10
  - Data dependences
    - Flow dependences
    - Antidependences
    - Output dependences
    - Input dependences

- Dependence graph
  - Control dependences are often converted to data-dependences
Dependence analysis

- What is a dependence?
  - A relationship between two computations
  - Places constraints on their execution order

- Two kinds of dependences

- Control dependences
  - 1: if (a == 3)
  - 2: b = u10

- Data dependences
  - Flow dependences
  - Antidependences
  - Output dependences
  - Input dependences

- Dependence graph
  - Control dependences are often converted to data-dependences
Dependence analysis

- What is a dependence?
  - A relationship between two computations
  - Places constraints on their execution order

- Two kinds of dependences

- Control dependences
  - 1: if (a == 3)
  - 2: b = u10

- Data dependences
  - Flow dependences
  - Antidependences
  - Output dependences
  - Input dependences

Dependence graph

Control dependences are often converted to data-dependences
Dependence analysis

- What is a dependence?
  - A relationship between two computations
  - Places constraints on their execution order

- Two kinds of dependences
- Control dependences
  - 1: if (a == 3)
  - 2: b = u10

- Data dependences
  - Flow dependences
  - Antidependences
  - Output dependences
  - Input dependences

- Dependence graph
- Control dependences are often converted to data-dependences
Data dependences examples

▶ Flow dependences

3: a = c*10
4: d = 2*a + c

▶ Antidependences

5: e = f*4 + g
6: g = 2*h

▶ Output dependences

7: a = b*c
8: a = d + e

▶ Input dependences

An opportunity for optimizing data placement
Data dependences examples

▶ Flow dependences
  ▶ 3: \( a = c \times 10 \)
  4: \( d = 2 \times a + c \)

▶ Antidependences
Data dependences examples

- Flow dependences
  - 3: \( a = c \times 10 \)
  - 4: \( d = 2 \times a + c \)

- Antidependences
  - 5: \( e = f \times 4 + g \)
  - 6: \( g = 2 \times h \)

- Output dependences
Data dependences examples

- Flow dependences
  - 3: \( a = c \times 10 \)
  - 4: \( d = 2 \times a + c \)

- Antidependences
  - 5: \( e = f \times 4 + g \)
  - 6: \( g = 2 \times h \)

- Output dependences
  - 7: \( a = b \times c \)
  - 8: \( a = d + e \)

- Input dependences
Data dependences examples

- **Flow dependences**
  - 3: \( a = c \times 10 \)
  - 4: \( d = 2 \times a + c \)

- **Antidependences**
  - 5: \( e = f \times 4 + g \)
  - 6: \( g = 2 \times h \)

- **Output dependences**
  - 7: \( a = b \times c \)
  - 8: \( a = d + e \)

- **Input dependences**
  - An opportunity for optimizing data placement
Loop dependence analysis

Loop carried dependences

1: for i = 2 to n
2: a[i] = a[i] + c
3: b[i] = a[i-1] + b[i]
Loop dependence analysis

- Loop carried dependences
  1: for i = 2 to n
  2: a[i] = a[i] + c
  3: b[i] = a[i-1] + b[i]

- Distance vectors
  - Describe distances between iterations
  - May be different than the distance between array elements
  - Must be positive
Loop dependence analysis

- Loop carried dependences
  1. for i = 2 to n
  2. a[i] = a[i] + c
  3. b[i] = a[i-1] + b[i]

- Distance vectors
  - Describe distances between iterations
  - May be different than the distance between array elements
  - Must be positive

- Discovering loop-carried dependences
  - Proving independence can be very difficult
  - Most compilers use a simple set of heuristics
Loop dependence analysis

- Loop carried dependences
  - 1: for i = 2 to n
  - 2: a[i] = a[i] + c
  - 3: b[i] = a[i-1] + b[i]

- Distance vectors
  - Describe distances between iterations
  - May be different than the distance between array elements
  - Must be positive

- Discovering loop-carried dependences
  - Proving independence can be very difficult
  - Most compilers use a simple set of heuristics

- When subscript expressions are too complex
  - The optimizer gives up
  - Statements are assumed to be fully dependent
Dataflow-based loop transformations

- Loop-based strength reduction
  - Replace operations with equivalent but less expensive ones

- Loop-invariant code motion
  - Sometimes expressions are constant within a loop
  - We can move that computation outside the loop
  - Caveat: Increases register pressure

- Loop unswitching
  - Loops often contain conditionals
  - If their conditions are loop-invariant they can be moved outside
Dataflow-based loop transformations

- Loop-based strength reduction
  - Replace operations with equivalent but less expensive ones
- Loop-invariant code motion
  - Sometimes expressions are constant within a loop
  - We can move that computation outside the loop
  - Caveat: Increases register pressure
Dataflow-based loop transformations

- Loop-based strength reduction
  - Replace operations with equivalent but less expensive ones
- Loop-invariant code motion
  - Sometimes expressions are constant within a loop
  - We can move that computation outside the loop
  - Caveat: Increases register pressure
- Loop unswitching
  - Loops often contain conditionals
  - If their conditions are loop-invariant they can be moved outside
Loop reordering

- Change the relative order of nested loops
Loop reordering

- Change the relative order of nested loops
  - Expose parallelism
  - Improve memory locality
- Techniques used
Loop reordering

- Change the relative order of nested loops
  - Expose parallelism
  - Improve memory locality

- Techniques used
  - Loop interchange
  - Loop skewing
  - Loop reversal
  - Strip mining
  - Cycle Shrinking
  - Loop tiling
  - Loop distribution
  - Loop fusion
Loop reordering

- Change the relative order of nested loops
  - Expose parallelism
  - Improve memory locality
- Techniques used
  - Loop interchange: Reduce stride
  - Loop skewing: Expose parallelism
  - Loop reversal: Reduce loop overhead
  - Strip mining: SIMD
  - Cycle Shrinking: Expose fine-grained parallelism
  - Loop tiling: Improve processor, register, TLB, page locality
  - Loop distribution: Create smaller lighter loops
  - Loop fusion: Reduce loop overhead
Loop restructuring

- Loop unrolling

- Very well known
- Very effective
- Reduces loop overhead
- Increases instruction level parallelism
- Improves locality
- Caveat: Increases code size

- Software pipelining
- Loop coalescing
- Combine a loop nest into a single loop
- Loop collapsing
- More efficient but less general than coalescing
- Loop peeling: Helps expose other optimizations
Loop restructuring

- Loop unrolling
  - Very well known
  - Very effective
  - Reduces loop overhead
  - Increases instruction level parallelism
  - Improves locality
  - Caveat: Increases code size
Loop restructuring

- Loop unrolling
  - Very well known
  - Very effective
  - Reduces loop overhead
  - Increases instruction level parallelism
  - Improves locality
  - Caveat: Increases code size

- Software pipelining
Loop restructuring

- Loop unrolling
  - Very well known
  - Very effective
  - Reduces loop overhead
  - Increases instruction level parallelism
  - Improves locality
  - Caveat: Increases code size

- Software pipelining

- Loop coalescing
  - Combine a loop nest into a single loop
Loop restructuring

- Loop unrolling
  - Very well known
  - Very effective
  - Reduces loop overhead
  - Increases instruction level parallelism
  - Improves locality
  - Caveat: Increases code size

- Software pipelining

- Loop coalescing
  - Combine a loop nest into a single loop

- Loop collapsing
  - More efficient but less general than coalescing
Loop restructuring

- Loop unrolling
  - Very well known
  - Very effective
  - Reduces loop overhead
  - Increases instruction level parallelism
  - Improves locality
  - Caveat: Increases code size

- Software pipelining

- Loop coalescing
  - Combine a loop nest into a single loop

- Loop collapsing
  - More efficient but less general than coalescing

- Loop peeling: Helps expose other optimizations
Loop replacement

- Reduction recognition
  - Compute a scalar from an array
  - For example: \textit{sum}, \textit{max}, or
  - Can be parallelized for commutative operations
Loop replacement

- Reduction recognition
  - Compute a scalar from an array
  - For example: sum, max, or
  - Can be parallelized for commutative operations
- Loop idiom recognition
  - Take advantage of SIMD hardware
Memory access transformations

- More and more applications become memory limited
- Substitute “memory” with “I/O” if you are DB oriented
Memory access transformations

- More and more applications become memory limited
- Substitute “memory” with “I/O” if you are DB oriented
- Popular techniques:
  - Array padding
  - Scalar expansion
  - Array contraction
  - Scalar replacement
  - Code collocation
  - Displacement minimization
Memory access transformations

- More and more applications become memory limited
- Substitute “memory” with “I/O” if you are DB oriented
- Popular techniques:
  - Array padding: reduces conflicts
  - Scalar expansion: help parallelize loops
  - Array contraction: reduce temporary storage
  - Scalar replacement: reduce frequent access overhead
  - Code collocation: improve memory access behavior
  - Displacement minimization: reduce jump distance
Partial evaluation

- Perform part of the computation at compile time
Partial evaluation

- Perform part of the computation at compile time
- Popular techniques:
  - Constant propagation
  - Constant folding
  - Copy propagation
  - Forward substitution
  - Reassociation
  - Algebraic simplification
  - Strength reduction
  - I/O format compilation
  - Superoptimizing
Redundancy elimination

- Remove redundant computations
Redundancy elimination

- Remove redundant computations
- Popular techniques:
  - Unreachable-code elimination
  - Useless-code elimination
  - Dead-variable elimination
  - Common-subexpression elimination
  - Short circuiting
To be continued...

- Thank you for your attention
- Questions?