Multi-Level Intermediate Representations

Local CSE, DAGs, Lowering Call Sequences

Survey of some Compiler Frameworks

Multi-Level IR

- Multi-level IR, e.g.
  - AST abstract syntax tree – implicit control and data flow
  - HIR high-level IR
  - MIR medium-level IR
  - LIR low-level IR, symbolic registers
  - VLIR very low-level IR, target specific, target registers
- Standard form and possibly also SSA (static single assignment) form
- Open form (tree, graph) and/or closed (linearized, flattened) form
  - For expressions: Trees vs DAGs (directed acyclic graphs)
- Translation by lowering
  - Analysis / Optimization engines can work on the most appropriate level of abstraction
  - Clean separation of compiler phases, somewhat easier to extend and debug
  - Framework gets larger and slower

Example: WHIRL

(Open64 Compiler)

Very High WHIRL (AST)
High WHIRL
Mid WHIRL
Low WHIRL
Very Low WHIRL

CGIR
CG

code generation, including scheduling, profiling support, predication, SW speculation

I/O

Very Low WHIRL
**Hierarchical symbol table follows nesting of scopes**

**Symbol table**

- Some typical fields in a symbol table entry

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char *</td>
<td>the symbol’s identifier</td>
</tr>
<tr>
<td>sclass</td>
<td>enum { STATIC, ...}</td>
<td>storage class</td>
</tr>
<tr>
<td>size</td>
<td>int</td>
<td>size in bytes</td>
</tr>
<tr>
<td>type</td>
<td>struct type *</td>
<td>source language data type</td>
</tr>
<tr>
<td>basetype</td>
<td>struct type *</td>
<td>source-lang. type of elements of a constructed type</td>
</tr>
<tr>
<td>machtype</td>
<td>enum {...}</td>
<td>machine type corresponding to source type (or element type if constructed type)</td>
</tr>
<tr>
<td>basereg</td>
<td>char *</td>
<td>base register to compute address</td>
</tr>
<tr>
<td>disp</td>
<td>int</td>
<td>displacement to address on stack</td>
</tr>
<tr>
<td>reg</td>
<td>char *</td>
<td>name of register containing the symbol’s value</td>
</tr>
</tbody>
</table>

**HIR - high-level intermediate representation**

- A (linearized) control flow graph, but level of abstraction close to AST
  - loop structures and bounds explicit
  - array subscripts explicit
  - suitable for data dependence analysis and loop transformation / parallelization

**Multi-Level IR Overview**

**Flattening 0:** From AST to HIR (or other CFG repr.)
Generating a CFG from AST
- Straightforward for structured programming languages
  - Traverse AST and compose control flow graph recursively
  - As in syntax-directed translation, but separate pass
  - Stitching points: single entry, single exit point of control;
    symbolic labels for linearization

Creating a CFG from AST (2)
- Traverse AST recursively, compose CFG
- Example:

    ```
    entry
    block
    if (condition)
    block
    block
    exit
    ```

HIR/MIR/LIR Example (adapted from Muchnick’97)
- HIR:
  ```
  for v = v1 by v2 to v3 do
    a[i] = 2
  endfor
  ```
- MIR:
  ```
  v = v1
  i2 = v2
  i3 = v3
  L1: if v > i3 goto L2
  i4 = addr a
  i5 = i4 + i5
  *i5 = 2
  v = v + i2
  goto L1
  ```
- LIR:
  ```
  s2 = s1
  s4 = s3
  s6 = s5
  L1: if s2 > s6 goto L2
  s7 = addr a
  s8 = 4 * s9
  *s10 = 2
  s2 = s2 + s4
  goto L1
  ```

Example with SSA-LIR (adapted from Muchnick’97)
- SSA-form makes data flow (esp., def-use chains) explicit
- Certain program analyses and transformations are easier to implement or more efficient on SSA-representation
- (Up to now) SSA is not suitable for code generation
- Requires transformation back to standard form
- Comes later...

SSA-Form vs. Standard Form of IR
- SSA form makes data flow (esp., def-use chains) explicit
- Certain program analyses and transformations are easier to implement or more efficient on SSA-representation
- (Up to now) SSA is not suitable for code generation
- Requires transformation back to standard form
- Comes later...

Multi-Level IR Overview
Standard vs. SSA Form

HIR
SSA-HIR
AST
HIR
SSA-HIR
LIR
SSA-LIR
VLIR (target code)
MIR – medium-level intermediate representation

- "language independent"
- control flow reduced to simple branches, call, return
- variable accesses still in terms of symbol table names
- explicit code for procedure / block entry / exit
- suitable for most optimizations
- basis for code generation

HIR→MIR (1): Flattening the expressions

By a postorder traversal of each expression tree in the CFG:
- Decompose the nodes of the expression trees (operators, ...) into simple operations (ADD, SUB, MUL, ...)
- Infer the types of operands and results (language semantics)
  - annotate each operation by its (result) type
  - insert explicit conversion operations where necessary
- Flatten each expression tree (= partial order of evaluation) to a sequence of operations (= total order of evaluation) using temporary variables t1, t2, ... to keep track of data flow
  - This is static scheduling!
  - May have an impact on space / time requirements

HIR→MIR (2): Lowering Array References (1)

HIR:
- t1 = a [i, j+2]

the Lvalue of a [i, j+2] is
(on a 32-bit architecture)
- (addr a) + 4 * (i * 20 + j + 2)

MIR:
- t1 = j + 2
- t2 = i * 20
- t3 = t1 + t2
- t4 = 4 * t3
- t5 = addr a
- t6 = t5 + t4
- t7 = *t6

HIR→MIR (2): Flattening the control flow graph

- Depth-first search of the control flow graph
- Topological ordering of the operations, starting with entry node
  - at conditional branches:
    - one exit fall-through, other exit branch to a label
- Basic blocks = maximum-length subsequences of statements containing no branch nor join of control flow
- Basic block graph obtained from CFG by merging statements in a basic block to a single node

Control flow graph

- Nodes: primitive operations (e.g., quadruples)
- Edges: control flow transitions

Example:

1: (JEQZ, 5, 0, 0)
2: (ASGN, 2, 0, A)
3: (ADD, A, 3, B)
4: (JUMP, 7, 0, 0)
5: (ASGN, 23, 0, A)
6: (SUB, A, 1, B)
7: (MUL, A, B, C)
8: (ADD, C, 1, A)
9: (JNEZ, B, 2, 0)
Basic block

- A basic block is a sequence of textually consecutive
  operations (e.g. MIR operations, LIR operations, quadruples)
  that contains no branches (except perhaps its last operation)
  and no branch targets (except perhaps its first operation).
  - Always executed in same order from entry to exit
  - A.k.a. straight-line code

Basic block graph

- Nodes: basic blocks
- Edges: control flow transitions

LIR – low-level intermediate representation

- in GCC: Register-transfer language (RTL)
- usually architecture dependent
  - e.g. equivalents of target instructions + addressing modes
    for IR operations
  - variable accesses in terms of target memory addresses

MIR→LIR: Lowering Variable Accesses

 Seen earlier:
- HIR: \[ t_1 = a \[ i, j+2 \] \]
- the Lvalue of \( a \[ i, j+2 \] \) is
  \[ \text{Memory layout:} \]
  - Local variables relative to
    procedure frame pointer fp
  - \( i \) at \( \text{fp} - 4 \)
  - \( j \) at \( \text{fp} - 8 \)
  - \( a \) at \( \text{fp} - 216 \)
- \[ \text{LIR:} \]
  \[ r_1 = [\text{fp} - 4] \]
  \[ r_2 = r_1 + 2 \]
  \[ r_3 = [\text{fp} - 8] \]
  \[ r_4 = r_3 + 20 \]
  \[ r_5 = r_4 + r_2 \]
  \[ r_6 = 4 \times r_5 \]
  \[ r_7 = \text{fp} - 216 \]
  \[ f_1 = [r_7 + r_6] \]

Example: The LCC-IR

- LIR – DAGs (Fraser, Hanson ’95)

Flattening 2: From MIR to LIR

- AST → HIR → SSA-HIR
- HIR → MIR → SSA-MIR
- MIR → LIR → SSA-LIR
- LIR → VLIR (target code)
MIR→LIR: Storage Binding
- mapping variables (symbol table items) to addresses
- (virtual) register allocation
- procedure frame layout implies addressing of formal parameters and local variables relative to frame pointer fp, and parameter passing (call sequences)
- for accesses, generate Load and Store operations
- further lowering of the program representation

MIR→LIR translation example

MIR:
\[
\begin{align*}
a &= a * 2 \\
b &= a + c \{1\}
\end{align*}
\]
LIR, bound to storage locations:
\[
\begin{align*}
r1 &= \text{[gp+8]} \text{ // Load} \\
r2 &= r1 * 2 \\
r3 &= \text{[gp+8]} \text{ // store} \\
r4 &= \text{[fp – 56]} \\
r5 &= r3 + r4 \\
\text{[fp – 20]} &= r5
\end{align*}
\]
LIR, bound to symbolic registers:
\[
\begin{align*}
s1 &= s1 * 2 \\
s2 &= \text{[fp – 56]} \\
s3 &= s1 + s2
\end{align*}
\]
Storage layout:
- Global variable a addressed relative to global pointer gp
- local variables b, c relative to fp

MIR→LIR: Procedure call sequence (0)

[Muchnick 5.6]

MIR call operation
- Call preparation (LIR code)
- Call instruction (LIR)
- Procedure prologue (LIR)
- Procedure epilogue (LIR)
- Return instruction (LIR)

MIR→LIR: Procedure call sequence (1)

[Muchnick 5.6]

MIR call instruction assembles arguments and transfers control to callee:
- evaluate each argument (reference vs. value param.) and
  - push it on the stack, or
  - write it to a parameter register
- determine code address of the callee (mostly, compile-time or link-time constant)
- store caller-save registers (usually, push on the stack)
- save return address (usually in a register) and branch to code entry of callee.

MIR→LIR: Procedure call sequence (2)

Procedure prologue
executed on entry to the procedure
- save old frame pointer fp
- old stack pointer sp becomes new frame pointer fp
- determine new sp (creating space for local variables)
- save callee-save registers

MIR→LIR: Procedure call sequence (3)

Procedure epilogue
executed at return from procedure
- restore callee-save registers
- put return value (if existing) in appropriate place (reg/stack)
- restore old values for sp and fp
- branch to return address

Caller cleans up upon return:
- restore caller-save registers
- use the return value (if applicable)
**From Trees to DAGs:**

**Common Subexpression Elimination (CSE)**

E.g., at MIR→LIR Lowering

---

**Local CSE on MIR produces a MIR DAG**

1. \( c = a \)
2. \( b = a + 1 \)
3. \( c = 2 \ast a \)
4. \( d = -c \)
5. \( c = a + 1 \)
6. \( c = b + a \)
7. \( d = 2 \ast a \)
8. \( b = c \)

---

**Flattening 3:**

**From LIR to VLIR**

---

**LIR→VLIR: Instruction selection**

- LIR has often a lower level of abstraction than most target machine instructions (esp., CISC, or DSP-MAC).
- One-to-one translation LIR-operation to equivalent target instruction(s) ("macro expansion") cannot make use of more sophisticated instructions
- Pattern matching necessary!

**LIR / VLIR: Register Allocation**

- Example for a SPARC-specific VLIR

```plaintext
int a, b, c, d;
khw a, r1
khw b, r2
add r1, r2, r3
stw r3, add c
khw add c, r3
add r3, r1, r4
stw r4, add d
```

---

There is a lot to be gained by good register allocation!
On LIR/VLIR: Global register allocation

- **Register allocation**
  - determine what values to keep in a register
  - “symbolic registers”, “virtual registers”
- **Register assignment**
  - assign virtual to physical registers
  - Two values cannot be mapped to the same register if they are **alive** simultaneously, i.e. their live ranges overlap (depends on schedule).

Remarks on IR design (1) [Cooper’02]

- Level of abstraction is critical for implementation cost and opportunities:
  - representation chosen affects the entire compiler

Example 1: Addressing for arrays and aggregates (structs)

- source level AST: hides entire address computation \(A[i+1][j]\)
- pointer formulation: may hide critical knowledge (bounds)
- low-level code: may make it hard to see the reference
  - “best” representation depends on how it is used
    - for dependence-based transformations: source-level IR (AST, HIR)
    - for fast execution: pointer formulation (MIR, LIR)
    - for optimizing address computation: low-level repr. (LIR, VLIR, target)

Remarks on IR Design (2)

**Example 2**: Representation for comparison & branch

- fundamentally, 3 different operations:
  - Compare \(\rightarrow\) convert result to boolean \(\rightarrow\) branch
  - combined in different ways by processor architects
  - “best” representation may depend on target machine

- \(r7 = (x < y)\) cmp x y (sets CC) \(r7 = r7\)
- \(br r7, L12\) brLT L12 \([r7]\) br L12

- “design problem for a retargetable compiler

Summary

- **Multi-level IR**
  - Translation by lowering
  - Program analyses and transformations can work on the most appropriate level of abstraction
  - Clean separation of compiler phases
  - Compiler framework gets larger and slower

Lowering:

- \(\text{AST} \downarrow\text{HIR} \rightarrow \text{SSA-HIR}\)
- \(\text{MIR} \downarrow\text{SSA-MIR}\)
- \(\text{LIR} \downarrow\text{SSA-LIR}\)
- \(\text{VLIR (target code)}\)

APPENDIX – For Self-Study

**Compiler Frameworks**

A (non-exhaustive) survey

with a focus on open-source frameworks
**LCC (Little C Compiler)**
- Dragon-book style C compiler implementation in C
- Very small (20K Loc), well documented, well tested, widely used
- Open source: http://www.cs.princeton.edu/software/lcc
- Textbook: A retargetable C compiler [Fraser, Hanson 1995]
- contains complete source code
- One-pass compiler, fast
- C frontend (hand-crafted scanner and recursive descent parser) with own C preprocessor
- Low-level IR
  - Basic-block graph containing DAGs of quadruples
- No AST
- Interface to IBURG code generator generator
- Example code generators for MIPS, SPARC, Alpha, x86 processors
- Tree pattern matching + dynamic programming
- Few optimizations only
- local common subexpr. elimination, constant folding
- Good choice for source-to-target compiling if a prototype is needed soon

**GCC 4.x**
- Gnu Compiler Collection (earlier: Gnu C Compiler)
- Compilers for C, C++, Fortran, Java, Objective-C, Ada ... (sometimes with own extensions, e.g. Gnu-C)
- Open-source, developed since 1985
- Very large
- 3 IR formats (all language independent)
  - GENERIC: tree representation for whole function (also statements)
  - GIMPLE: simple version of GENERIC for optimizations
  - based on trees but expressions in quadruple form
  - High-level, low-level and SSA-low-level form
- RTL (Register Transfer Language, low-level, Lisp-like) (the traditional GCC-IR)
- only word-sized data types; stack explicit; statement scope
- Many optimizations
- Many target architectures
- Version 4.x (since ~2004) has strong support for retargetable code generation
  - Machine description in .md file
  - Reservation tables for instruction scheduler generation
  - Good choice if one has the time to get into the framework

**Open64 / ORC Open Research Compiler**
- Based on SGI Pro-64 Compiler for MIPS processor, written in C++, went open source in 2000
- Several tracks of development (Open64, ORC, …)
- For Intel Itanium (IA-64) and x86 (IA-32) processors. Also retargeted to x86-64, Ceva DSP, Tensilica, XScale, ARM ...
  - “simple to retarget” (?)
- Languages: C, C++, Fortran95 (uses GCC as frontend), OpenMP and UPC (for parallel programming)
- Industrial strength, with contributions from Intel, Pathscale, ...
- Open source: www.open64.net, ipf-orc.sourceforge.net
- 6-layer IR:
  - WHIRL (VH, H, M, L, VL) – 5 levels of abstraction
    - All levels semantically equivalent
    - Each level a lower level subset of the higher form
    - and target-specific very low-level CGIR
- Many optimizations, many third-party contributed components

**LLVM**
- (llvm.org)
- LLVM (Univ. of Illinois at Urbana Champaign)
  - “Low-level virtual machine”
  - Front-ends (Clang, GCC) for C, C++, Objective-C, Fortran, ...
  - One IR level: a LIR + SSA-LIR,
    - linearized form, printable, shippable, but target-dependent,
    - LLVM instruction set
  - compiles to many target platforms
    - x86, Itanium, ARM, Alpha, SPARC, PowerPC, Cell SPE, ...
  - And to low-level C
  - Link-time interprocedural analysis and optimization framework
    - for whole-program analysis
  - JIT support available for x86, PowerPC
  - Open source

**Open64 WHIRL**
- VHO (standalone inliner)
- IPA (interprocedural analysis)
- PREOPT
- LNO (Loop nest optimizer)
- WOPT (global optimizer, uses internally an SSA (IR))
- RVI1 (register variable identification)
- RVI2 (code generation)
- CG (code generation, including scheduling, profiling support, predication, SW speculation)
- CGIR (LLVM IR similar to CGIR in LCC)
- Low WHIRL
  - Front-ends (GCC)
  - Lower aggregates
  - Lower arrays
  - Lower complex numbers
  - Lower HL control flow
  - Lower bit-fields
  - Lower intrinsic ops to calls
  - All data mapped to segments
  - Expose code sequences for constants, addresses
  - Expose # (gp) addr. for globals

**VEX Compiler**
- VEX: "VLIW EXample"
  - Generic clustered VLIW Architecture and Instruction Set
- From the book by Fisher, Faraboschi, Young: Embedded Computing, Morgan Kaufmann 2005
- Developed at HP Research
  - Based on the compiler for HP/ST Lx (ST200 DSP)
- Compiler, Libraries, Simulator and Tools
  - available in binary form from HP for non-commercial use
  - IR not accessible, but CFGs and DAGs can be dumped or visualized
  - Transformations controllable by options and/or # pragmas
  - Scalar optimizations, loop unrolling, prefetching, function inlining, ...
  - Global scheduling (esp., trace scheduling), but no software pipelining

**References**
- Fraser, Hanson 1995
- www.open64.org
- www.llvm.org
- C. Kessler, IDA, Linköpings universitet.

---

**Embedd**ed Computing, Morgan Kaufmann 2005

---

This document provides an overview of various compiler projects and technologies, including LCC (Little C Compiler), GCC 4.x, Open64 / ORC Open Research Compiler, LLVM, Open64 WHIRL, and VEX Compiler. Each project is described in detail, highlighting its features, optimizations, and target architectures. The document also references key authors and sources for further reading.
A commercial compiler framework

www.ace.nl

Traditional Compiler Structure

- Traditional compiler model: sequential process
- Improvement: Pipelining (by files/modules, classes, functions)
- More modern compiler model with shared symbol table and IR:

A CoSy Compiler with Repository-Architecture

Engine

- Modular compiler building block
- Performs a well-defined task
- Focus on algorithms, not compiler configuration
- Parameters are handles on the underlying common IR repository
- Execution may be in a separate process or as subroutine call - the engine writer does not know!
- View of an engine class: the part of the common IR repository that it can access (scope set by access rights: read, write, create)
- Examples: Analyzers, Lowerers, Optimizers, Translators, Support

Composite Engines in CoSy

- Built from simple engines or from other composite engines by combining engines in interaction schemes (Loop, Pipeline, Fork, Parallel, Speculative, …)
- Described in EDL (Engine Description Language)
- View defined by the joint effect of constituent engines
- A compiler is nothing more than a large composite engine
Example for CoSy EDL (Engine Description Language)

Component classes (engine class)
Component instances (engines)
Basic components are implemented in C
Interaction schemes (cf. skeletons) form complex connectors
  SEQUENTIAL
  PIPELINE
  DATAPARALLEL
  SPECULATIVE
EDL can embed automatically
  Single-call-components into pipes
  p<> means a stream of p-items
EDL can map their protocols to each other (p vs p<>)

Example for CoSy EDL (Engine Description Language)

Component classes (engine class)
Component instances (engines)
Basic components are implemented in C
Interaction schemes (cf. skeletons) form complex connectors
  SEQUENTIAL
  PIPELINE
  DATAPARALLEL
  SPECULATIVE
EDL can embed automatically
  Single-call-components into pipes
  p<> means a stream of p-items
EDL can map their protocols to each other (p vs p<>)

Engine Class optimizer (procedure p)
  ControlFlowAnalyser cfa;
  CommonSubExprEliminator cse;
  LoopVariableSimplifier lvs;
  PIPELINE cfa(p); cse(p); lvs(p);

Engine Class compiler (file f)
  Token token;
  Module m;
  PIPELINE
  lexer( IN f, OUT token<> );
  parser( IN token<>, OUT m );
  sema( m );
  decompose( m, p<> );
  // here comes a stream of procedures
  // from the module
  optimizer( p<> );
  backend( p<> );

Evaluation of CoSy

- The outer call layers of the compiler are generated from view description specifications
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged
  - There is also a non-commercial prototype

- Access layer to the repository must be efficient (solved by generation of macros)
- Because of views, a CoSy-compiler is very simply extensible
  - That's why it is expensive
  - Reconfiguration of a compiler within an hour

Source-to-Source compiler frameworks

- Cetus
  - C / OpenMP source-to-source compiler written in Java.
  - Open source
- ROSE
  - C++ source-to-source compiler
  - Open source
- Tools and generators
  - TXL source-to-source transformation system
  - ANTLR frontend generator

More frameworks (mostly historical) ...

- Some influential frameworks of the 1990s ...some of them still active today
  - SUIF Stanford university intermediate format, suif.stanford.edu
  - Trimaran (for instruction-level parallel processors) www.trimaran.org
  - Polaris (Fortran) UIUC
  - Jikes RVM (Java) IBM
  - Soot (Java)
  - GMD Toolbox / Cocolab Cocktail™ compiler generation tool suite
  - and many others ...
- And many more for the embedded domain ...