Architecture for a Next-Generation GCC

Chris Lattner
sabre@nondot.org

Vikram Adve
vadve@cs.uiuc.edu

http://llvm.cs.uiuc.edu/

The First Annual GCC Developers' Summit
May 26, 2003
GCC Optimizer Problems:

- **Scope of optimization is very limited**:  
  - Most transformations work on functions…  
  - …and one is even limited to extended basic blocks  
  - No *whole-program* analyses or optimization!  
  - e.g. alias analysis must be extremely conservative

- **Tree & RTL are bad for mid-level opt’zns**:  
  - Tree is language-specific and too *high*-level  
  - RTL is target-specific and too *low*-level
New Optimization Architecture:

- **Transparent** *link-time* optimization:
  - Completely compatible with user makefiles

- Enables sophisticated interprocedural analyses (IPA) and optimizations (IPO):
  - Increase the scope of analysis and optimization

- **A new representation for optimization:**
  - Typed, SSA-based, three-address code
  - Source language *and* target-independent
Example Applications for GCC:

- **Fix inlining heuristics:**
  - Allows whole program, bottom-up inlining
  - Cost metric is more accurate than for trees

- **Improved alias analysis:**
  - Dramatically improved precision
  - Code motion, redundancy elimination gains

- **Work around low-level ABI problems:**
  - Tailor linkage of functions with IP information
Talk Outline:

- **High-Level Compiler Architecture**
  - How does the proposed GCC work?

- **Code Representation Details**
  - What does the representation look like?

- **LLVM: An Implementation**
  - Implementation status and experiences

- **Conclusion**
Traditional GCC Organization:

☞ **Compile:** source to target assembly
☞ **Assemble:** target assembly to object file
☞ **Link:** combine object files into an executable
Proposed GCC Architecture:

Split the existing compiler in half:

- Parsing & semantic analysis at compile time
- Code generation at link-time
- Optimization at compile-time *and* link-time
Why Link-Time?

Fits into normal compile & link model:
- User makefiles do not have to change
- Enabled if compiling at `-O4`

Missing code severely limits IPA & IPO:
- Must make conservative assumptions:
  - An unknown callee can do just about anything
- At link-time, most of the program is available for the first time!
Making Link-Time Opt Feasible:

- Many commercial compilers support link-time optimization (Intel, SGI, HP, etc...):
  - These export an AST-level representation, then perform **all** optimization at link-time

- Our proposal:
  - Optimize as much at **compile-time** as possible
  - Perform aggressive IPA/IPO at link-time
  - Allows mixed object files in native & IR format
No major GCC changes:

**New GCC components:**
- New expander from Tree to IR
- New expander from IR to RTL
- Must extend the compiler driver

**Existing code path can be retained:**
- When disabled, does not effect performance
- When `-O2` is enabled, use new mid-level optimizations a function- (or unit-) at-a-time
Talk Outline:

- High-Level Compiler Architecture
  - How does the proposed GCC work?
- Code Representation Details
  - What does the representation look like?
- LLVM: An Implementation
  - Implementation status and experiences
- Conclusion
Code Representation Properties:

- Low-Level, SSA based, and “RISC-like”:
  - SSA-based = efficient, sparse, global opt’zns
  - Orthogonal, as few operations as possible
  - Simple, well defined semantics (documented)
  - Simplify development of optimizations:
    - Development & maintenance is very costly!

- Concrete details come from LLVM:
  - More details about LLVM come later in talk
Code Example:

```c
struct pair {
    int X; float Y;
};
void Sum(float *, void Sum(float *, struct pair *P);

int int Process(float *A, int int N) {
    int i;
    struct pair P = {0, 0};
    for (i = 0; i < N; ++i) {
        Sum(A, &P);
        A++; }   A++; }
return P.X;
```
Strongly-Typed Representation:

Key challenge:
- Support high-level analyses & transformations
- ... on a low-level representation!

Types provide this high-level info:
- Enables aggressive analyses and opt’zns:
  - e.g. automatic pool allocation, safety checking, data structure analysis, etc…
- Every computed value has a type

Type system is language-neutral!
Type System Details:

Simple lang. independent type system:
- Primitives: void, bool, float, ushort, opaque, ...
- Derived: pointer, array, structure, function
- No high-level types!

Source language types are lowered:
- e.g. T& ↳ T*
- e.g. class T : S { int X; } ↳ { S, int }

Type system can be “broken” with casts
Full Featured Language:

- Should contain all info about the code:
  - functions, globals, inline asm, etc…
  - Should be possible to serialize and deserialize a program at any time

- Language has binary and text formats:
  - Both directly correspond to in-memory IR
  - Text is for humans, binary is faster to parse
  - Makes debugging and understanding easier!
Talk Outline:

- High-Level Compiler Architecture
  - How does the proposed GCC work?
- Code Representation Details
  - What does the representation look like?
- LLVM: An Implementation
  - Implementation status and experiences
- Conclusion
LLVM: Low-Level Virtual Machine

A research compiler infrastructure:
- Provides a solid foundation for research
- In use both inside and outside of UIUC:
  - Compilers, architecture, & dynamic compilation
  - Two advanced compilers courses

Development Progress:
- 2.5 years old, ~130K lines of C++ code
- First public release is coming soon:
  - 1.0 release this summer, prereleases via email

Chris Lattner – sabre@nondot.org
LLVM Implementation Status:

Most of this proposal is implemented:

- Tree
- LLVM expander (for C and C++)
- Linker, optimizer, textual & bytecode formats
- Mid-level optimizer is sequence of 22 passes

All sorts of analyses & optimizations:

- Scalar: ADCE, SCCP, register promotion, …
- CFG: dominators, natural loops, profiling, …
- IP: alias analysis, automatic pool allocation, interprocedural mod/ref, safety verification…
Other LLVM Infrastructure:

- **Direct execution of LLVM bytecode:**
  - A portable interpreter, a Just-In-Time compiler

- **Several custom (non-GCC) backends:**
  - Sparc-V9, IA-32, C backend

- **The LLVM “Pass Manager”:**
  - Declarative system for tracking analysis and optimizer pass dependencies
  - Assists building tools out of a series of passes
LLVM Development Tools:

Invariant checking:
- Automatic IR memory leak detection
- A verifier pass which checks for consistency
  - Definitions dominate all uses, etc…

Bugpoint - automatic test-case reducer:
- Automatically reduces test cases to a small example which still causes a problem
- Can debug miscompilations or pass crashes
LLVM is extremely fast:

- End-to-end performance isn’t great yet:
  - Not yet integrated into GCC proper

- But transformations are very fast:

  Some example numbers from the paper:

<table>
<thead>
<tr>
<th>Source Filename</th>
<th>wc -l LOC</th>
<th>GCC CSE 1</th>
<th>LLVM Pass Times</th>
<th># LLVM Pass xforms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>IC</td>
<td>GER</td>
<td>GCSE</td>
</tr>
<tr>
<td>combine.c</td>
<td>11103</td>
<td>0.70s</td>
<td>.431s .027s .141s</td>
<td>.599s</td>
</tr>
<tr>
<td>expr.c</td>
<td>10747</td>
<td>0.52s</td>
<td>.141s .009s .072s</td>
<td>.222s</td>
</tr>
<tr>
<td>cse.c</td>
<td>8779</td>
<td>0.50s</td>
<td>.187s .012s .061s</td>
<td>.260s</td>
</tr>
<tr>
<td>reload1.c</td>
<td>7117</td>
<td>0.37s</td>
<td>.058s .008s .034s</td>
<td>.100s</td>
</tr>
<tr>
<td>c-decl.c</td>
<td>6968</td>
<td>0.42s</td>
<td>.022s .005s .031s</td>
<td>.058s</td>
</tr>
<tr>
<td>insn-recog.c</td>
<td>6957</td>
<td>0.34s</td>
<td>.082s .004s .090s</td>
<td>.176s</td>
</tr>
<tr>
<td>loop.c</td>
<td>6648</td>
<td>0.33s</td>
<td>.013s .001s .003s</td>
<td>.017s</td>
</tr>
<tr>
<td>c-typeck.c</td>
<td>6604</td>
<td>0.46s</td>
<td>.028s .005s .026s</td>
<td>.059s</td>
</tr>
</tbody>
</table>
Conclusion:

Contributions:

- A realistic architecture for an aggressive link-time optimizer
- A representation for efficient and powerful analyses and transformations

LLVM is available...

... and we appreciate your feedback!

http://llvm.cs.uiuc.edu

Chris Lattner – sabre@nondot.org