Modeling and Solving Code Generation for Real

Christian Schulte
KTH Royal Institute of Technology & SICS (Swedish Institute of Computer Science)

joint work with: Mats Carlsson SICS
                Roberto Castañeda Lozano SICS + KTH
                Frej Drejhammar SICS
                Gabriel Hjort Blindell KTH + SICS

funded by: Ericsson AB
           Swedish Research Council (VR 621-2011-6229)
Who Am I?

• Professor of Computer Science at KTH Royal Institute of Technology, Stockholm, Sweden
• Expert Researcher at SICS (Swedish Institute of Computer Science), Stockholm, Sweden

• Education
  • diploma in computer science, Karlsruhe, Germany, 1992
  • doctoral degree in engineering, Saarbrücken, Germany, 2001
  • docent in computer systems, KTH, Sweden, 2009

• Research interests
  • constraint programming
  • programming languages
  • systems-based research (Gecode, for example)
Compilation

- Front-end: depends on source programming language
  - changes infrequently (well...)

- Optimizer: independent optimizations
  - changes infrequently (well...)

- Back-end: depends on processor architecture
  - changes often: new process, new architectures, new features, ...
Generating Code: Unison

- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example

- Frequent changes: back-end
  - use flexible approach: Unison
State of the Art

- Code generation organized into stages
  - instruction selection,

\[ x = y + z; \]

\[
\begin{align*}
\text{add} & \ r0 \ r1 \ r2 \\
\text{mv} & \ $a6f0 \ r0
\end{align*}
\]
State of the Art

- Code generation organized into stages
  - instruction selection, register allocation,

\[
x = y + z;
\]

\[
x \rightarrow \text{register } r0
y \rightarrow \text{memory (spill to stack)}
\]

...
State of the Art

• Code generation organized into stages
  • instruction selection, register allocation, instruction scheduling

```
x = y + z;
...
u = v - w;
```

```
u = v - w;
...
x = y + z;
```
State of the Art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible
State of the Art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible

- Example: instruction scheduling ⇔ register allocation
  - increased delay between instructions can increase throughput
    → registers used over longer time-spans
    → more registers needed
State of the Art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible

- Example: instruction scheduling ⇨ register allocation
  - put variables into fewer registers
    → more dependencies among instructions
    → less opportunity for reordering instructions
State of the Art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible

- Stages use heuristic algorithms
  - for hard combinatorial problems (NP hard)
  - assumption: optimal solutions not possible anyway
  - difficult to take advantage of processor features
  - error-prone when adapting to change
State of the Art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algorithms
  - for hard combinatorial problems
  - assumption: optimal solutions not possible anyway
  - difficult to take advantage of processor features
  - error-prone when adapting
Rethinking: Unison Idea

- No more staging and complex heuristic algorithms!
  - many assumptions are decades old...

- Use state-of-the-art technology for solving combinatorial optimization problems: constraint programming
  - tremendous progress in last two decades...

- Generate and solve single model
  - captures all code generation tasks in unison
  - high-level of abstraction: based on processor description
  - flexible: ideally, just change processor description
  - potentially optimal: tradeoff between decisions accurately reflected
Unison Approach

- Generate constraint model
  - based on input program and processor description
  - constraints for all code generation tasks
  - generate but not solve: simpler and more expressive
Unison Approach

- Off-the-shelf constraint solver solves constraint model
  - solution is assembly program
  - optimization takes inter-dependencies into account
Talk Overview

- Constraint programming in a nutshell

- Register Allocation & Instruction Scheduling
  - Basic Register Allocation
  - Instruction Scheduling
  - Advanced Register Allocation
  - Global Register Allocation
  - Discussion

- Instruction Selection [if time allows]
  - Graph-based Instruction Selection
  - Universal Instruction Selection
  - Discussion

- Summary
Source Material

• Register Allocation & Instruction Scheduling
  • Constraint-based Register Allocation and Instruction Scheduling, Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, Christian Schulte. CP 2012.

• Instruction Selection
  • Modeling Universal Instruction Selection, Gabriel Hjort Blindell, Roberto Castañeda Lozano, Mats Carlsson, Christian Schulte. CP 2015.
Source Material

• Surveys
  
  

• Additional Material
  
  
CONSTRAINT PROGRAMMING IN A NUTSHELL
Constraint Programming

• Model and solve combinatorial (optimization) problems

• Modeling
  • variables
  • constraints
  • branching heuristics
  • (cost function)

• Solving
  • constraint propagation
  • heuristic search

• Of course simplified...
  ...array of modeling and solving techniques
Problem: Send More Money

• Find distinct digits for letters such that

\[
\begin{align*}
\text{SEND} & \quad + \quad \text{MORE} \\
\hline \\
\text{MONEY} & \quad =
\end{align*}
\]
Constraint Model

• Variables:
  \( S, E, N, D, M, O, R, Y \in \{0, \ldots, 9\} \)

• Constraints:
  \[
  \text{distinct}(S,E,N,D,M,O,R,Y) \\
  1000 \times S + 100 \times E + 10 \times N + D \\
  + 1000 \times M + 100 \times O + 10 \times R + E \\
  = 10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y \\
  S \neq 0 \quad M \neq 0
  \]
Constraints

- State relations between variables
  - legal combinations of values for variables

Examples
- all variables pair wise distinct: \(\text{distinct}(x_1, \ldots, x_n)\)
- arithmetic constraints: \(x + 2 \times y = z\)
- domain-specific: \(\text{cumulative}(t_1, \ldots, t_n)\)
  \(\text{nooverlap}(r_1, \ldots, r_n)\)

Success story: \textbf{global} constraints
- modeling: capture recurring problem structures
- solving: enable strong reasoning
  constraint-specific methods
Solving: Variables and Values

- Record possible values for variables
  - solution: single value left
  - failure: no values left

\[ x \in \{1,2,3,4\} \quad y \in \{1,2,3,4\} \quad z \in \{1,2,3,4\} \]
Constraint Propagation

- $\text{distinct}(x, y, z)$
- $x + y = 3$

$x \in \{1,2,3,4\}$  $y \in \{1,2,3,4\}$  $z \in \{1,2,3,4\}$

- Prune values that are in conflict with constraint
Constraint Propagation

- Prune values that are in conflict with constraint

\[
\text{distinct}(x, y, z) \quad x + y = 3
\]

\[
x \in \{1, 2\} \quad y \in \{1, 2\} \quad z \in \{1, 2, 3, 4\}
\]
Constraint Propagation

- Prune values that are in conflict with constraint
  - propagation is often smart if not perfect!

\[
\begin{align*}
\text{distinct}(x, y, z) & \quad x + y = 3 \\
\{1,2\} & \quad \{1,2\} \quad \{3,4\}
\end{align*}
\]
Heuristic Search

- Propagation *alone* not sufficient
  - decompose into simpler sub-problems
  - search needed
- Create subproblems with additional constraints
  - enables further propagation
  - defines *search tree*
  - uses problem specific heuristic
What Makes It Work?

• Essential: avoid search...
  ...as it always suffers from combinatorial explosion

• Constraint propagation drastically reduces search space

• Efficient and powerful methods for propagation available

• When using search, use a clever heuristic

• Array of modeling techniques available that reduce search

• Hybrid methods (together with LP, SAT, stochastic, ...)

Sep 10, 2015
Register Allocation & Instruction Scheduling
Unit and Scope

• Function is unit of compilation
  • generate code for one function at a time

• Scope
  • local  generate code for each basic block in isolation
  • global generate code for whole function

• Basic block: instructions that are always executed together
  • execute at start
  • execute all instructions
  • leave execution at end
  • that is: no control flow within basic block (in or out)
Local (and slightly naïve) register allocation

BASIC REGISTER ALLOCATION
Local Register Allocation

- Instruction selection has already been performed
- Temporaries
  - defined or def-occurrence (lhs) \( t_3 \) in \( t_3 \leftarrow \text{sub} \ t_1, 2 \)
  - used or use-occurrence (rhs) \( t_1 \) in \( t_3 \leftarrow \text{sub} \ t_1, 2 \)
- Basic blocks are in SSA (single static assignment) form
  - each temporary is defined once
  - standard state-of-the-art approach

\[

t_2 \leftarrow \text{mul} \ t_1, 2 \\
t_3 \leftarrow \text{sub} \ t_1, 2 \\
t_4 \leftarrow \text{add} \ t_2, t_3 \\
... \\
t_5 \leftarrow \text{mul} \ t_1, t_4 \\
\leftarrow \text{jr} \ t_5
\]
Liveness & Interference

- Temporary is **live** from def to last use, defining its **live range**
  - live ranges are **linear** (basic block + SSA)

- Temporaries **interfere** if their live ranges overlap

- Non-interfering temporaries can be assigned to same register
Spilling

• If not enough registers available: spill

• Spilling moves temporary to memory (stack)
  • store in memory after defined
  • load from memory before used
  • memory access typically considerably more expensive
  • decision on spilling crucial for performance

• Architectures might have more than one register bank
  • some instructions only capable of addressing a particular bank
  • “spilling” from one register bank to another

• **Unified register array**
  • limited number of registers for each register file
  • memory is just another “register” file
  • unlimited number of memory “registers”
Coalescing

- Temporaries $d$ ("destination") and $s$ ("source") are move-related if
  $$d \leftarrow s$$
  - $d$ and $s$ should be coalesced (assigned to same register)
  - coalescing saves move instructions and registers

- Coalescing is important due to
  - how registers are managed (calling convention)
  - how our model deals with global register allocation (more later)
Copy Operations

• Copy operations replicate a temporary $t$ to a temporary $t'$
  \[ t' \leftarrow \{i_1, i_2, \ldots, i_n\} t \]
  
  • copy is implemented by one of the alternative instructions $i_1, i_2, \ldots, i_n$
  
  • instruction depends on where $t$ and $t'$ are stored
    similar to [Appel & George, 2001]

• Example MIPS32
  \[ t' \leftarrow \{\text{move, sw, nop}\} t \]
  
  • $t'$ memory and $t$ register: \text{sw} spill
  
  • $t'$ register and $t$ register: \text{move} move-related
  
  • $t'$ and $t$ same register: \text{nop} coalescing
  
  • MIPS32: instructions can only be performed on registers
Model Variables

- **Decision variables**
  - \( \text{reg}(t) \in \mathbb{N} \) register to which temporary \( t \) is assigned
  - \( \text{instr}(o) \in \mathbb{N} \) instruction that implements operation \( o \)
  - \( \text{cycle}(o) \in \mathbb{N} \) issue cycle for operation \( o \)
  - \( \text{active}(o) \in \{0,1\} \) whether operation \( o \) is active

- **Derived variables**
  - \( \text{start}(t) \) start of live range of temporary \( t \)
    \[ = \text{cycle}(o) \quad \text{where } o \text{ defines } t \]
  - \( \text{end}(t) \) end of live range of temporary \( t \)
    \[ = \max \{ \text{cycle}(o) \mid o \text{ uses } t \} \]
Sanity Constraints

• Copy operation $o$ is active $\iff$ no coalescing
  \[ \text{active}(o) = 1 \iff \text{reg}(s) \neq \text{reg}(d) \]
  • $s$ is source of move, $d$ is destination of move operation $o$

• Operations implemented by suitable instructions
  • single possible instruction for non-copy operations

• Miscellaneous
  • some registers are pre-assigned
  • some instructions can only address certain registers (or memory)
Geometrical Interpretation

- Temporary $t$ is rectangle
  - width is 1 (occupies one register)
  - top = start($t$) issue cycle of def
  - bottom = end($t$) last issue cycle of any use

- Consequence of linear live range (basic block + SSA)
Register Assignment

- Register assignment = geometric packing problem
  - find horizontal coordinates for all temporaries
  - such that no two rectangles for temporaries overlap

- For block $B$

\[
\text{nooverlap}(\{\langle \text{reg}(t), \text{reg}(t)+1, \text{start}(t), \text{end}(t) \rangle \mid t \in B\})
\]
Register Packing

- Temporaries might have different width \( \text{width}(t) \)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
Register Packing

- Temporaries might have different width $width(t)$
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries ($width = 1$) to 16 bit register ($width = 2$)
  - register parts: $AH, AL, BH, BL, CH, CL$
  - possible for 8 bit: $AH, AL, BH, BL, CH, CL$
  - possible for 16 bit: $AH, BH, CH$
Register Packing

- Temporaries might have different width $\text{width}(t)$
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: $\text{AH, AL, BH, BL, CH, CL}$
  - possible for 8 bit: $\text{AH, AL, BH, BL, CH, CL}$
  - possible for 16 bit: $\text{AH, BH, CH}$
Register Packing

- Temporaries might have different width \( width(t) \)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: \( AH, AL, BH, BL, CH, CL \)
  - possible for 8 bit: \( AH, AL, BH, BL, CH, CL \)
  - possible for 16 bit: \( AH, BH, CH \)
Modeling Register Packing

- Take width of temporaries into account (for block $B$)
  \[
  \text{nooverlap} \{(\text{reg}(t), \text{reg}(t)+\text{width}(t), \text{start}(t), \text{end}(t)) \mid t \in B\}
  \]

- Exclude sub-registers depending on $\text{width}(t)$
  - simple domain constraint on $\text{reg}(t)$
Local instruction scheduling (standard)

INSTRUCTION SCHEDULING
Dependencies

- Data and control dependencies
  - data, control, artificial (for making in and out first/last)

- If operation $o_2$ depends on $o_1$:
  
  $\text{active}(o_1) \land \text{active}(o_2) \rightarrow$
  
  $\text{cycle}(o_2) \geq \text{cycle}(o_1) + \text{latency}(\text{instr}(o_1))$
Processor Resources

• Processor resources: functional units, data buses, ...
  • also: instruction bundle width for VLIW processors (how many instructions can be issued simultaneously)

• Classical cumulative scheduling problem
  functional units
  • processor resource has capacity #units
  • instructions occupy parts of resource 1 unit
  • resource consumption can never exceed capacity

• Modeling for block $B$
  \[
  \text{cumulative}\{(\langle\text{cycle}(o),\text{dur}(o,r),\text{active}(o)\times\text{use}(o,r)\rangle \mid o \in B)}\}
Ultimate Coalescing & Spill Code Optimization using alternative temporaries

ADVANCED REGISTER ALLOCATION
Interference Too Naïve!

- Move-related temporaries might interfere...
  ...but contain the same value!

- Ultimate notion of interference =
  temporaries interfere \( \iff \) their live ranges overlap and
  they have different values

[Chaitin ea, 1981]
Spilling Too Naïve!

- Known as **spill-everywhere** model
  - reload from memory before every use of original temporary

- Example: $t_3$ should be used rather than reloading $t_2$
  - $t_2$ allocated in memory!
Alternative Temporaries

• Used to track which temporaries are equal

• Representation is augmented by operands
  • act as def and use ports in operations
  • temporaries hold values transferred among operations by connecting to operands

• Example
  • operation $t_2 \leftarrow \text{abs } t_1$
  • transformed to $p_2:t_2 \leftarrow \text{abs } p_1:t_1$ ({$p_1, p_2$ operands})
  • if $t_1$ and $t_3$ hold same value then transformed to
    $p_2:t_2 \leftarrow \text{abs } p_1:{t_1,t_3}$
    where either $t_1$ or $t_3$ can be connected to $p_1$

• Model: whether a temporary is live (it is being used)
Register allocation for entire functions

GLOBAL REGISTER ALLOCATION
Entire Functions

int fac(int n) {
    int f = 1;
    while (n > 0) {
        f = f * n; n--;
    }
    return f;
}

• Use control flow graph (CFG) and turn it into LSSA form
  • edges = control flow
  • nodes = basic blocks (no control flow)
• LSSA = linear SSA = SSA for basic blocks plus... to be explained
Linear SSA (LSSA)

- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence \( t \equiv t' \iff \) represent same original temporary
Linear SSA (LSSA)

- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence \( \equiv \)
  \[ t \equiv t' \iff \text{represent same original temporary} \]
- Example: \( t_3, t_7, t_8, t_{11} \) are congruent
  - correspond to the program variable \( f \) (factorial result)
  - not discussed: \( t_1 \) return address, \( t_2 \) first argument, \( t_{11} \) return value
Linear SSA (LSSA)

- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence $t \equiv t'$ represent same original temporary
- Advantage
  - simple modeling for linear live ranges (geometrical interpretation)
  - enables problem decomposition for solving
Global Register Allocation

- Try to coalesce congruent temporaries
  - this is why coalescing is (even more) crucial in this model

- Introduces natural problem decomposition
  - master problem (function) coalesce congruent temporaries
  - slave problems (basic blocks) register allocation & instruction scheduling

- What is happening
  - if register pressure is low...
    no copy instruction needed (nop)
    = coalescing
  - if register pressure is high...
    copy operation might be implemented by a move
    = no coalescing
    copy operation might be implemented by a load/store
    = spill
DISCUSSION
Solving

• Approach
  • use master-slave decomposition
  • use naïve (very) portfolio of heuristics for basic blocks
  • use some pre-solving (symmetry, no-goods, dominance)
  • not very advanced (future work)

• Benchmark setup
  • selection of medium-sized functions (25 to 1000 instructions)
  • comparison to LLVM 3.3 for Qualcomm’s Hexagon V4 using –O3
  • run for ten iterations where each iteration is given more time
  • using Gecode 4.2.1
  • full details in [Castañeda ea, LCTES 2014]
Experiments Summary

• Code quality (estimated)
  • 7% mean improvement over LLVM
  • provably optimal for 29% of functions

• Quadratic average (roughly) complexity up to 1000 instructions

• Can be easily changed to optimize for code size
  • 1% mean improvement over LLVM
Related Approaches

• Idea and motivation in Unison for combinatorial optimization is absolutely not new!
  • starting in the early 1990s
    [Castañeda & Schulte, CoRR 2014]

• Approaches differ
  • which code generation tasks covered
  • which technology used (ILP, CP, SAT, Genetic Algorithms, ...)

• Common to most approaches
  • compilation unit is basic block, or
  • just a single task covered, or
  • very poor scalability

• Challenge: integration, robustness, and scalability
Unique to Unison Approach

• First global approach for register allocation (function as compilation unit)

• Constraint programming using global constraints
  • sweet spot: cumulative and no overlap

• Full register allocation with ultimate coalescing, packing, spilling, and spill code optimization
  • key property of model: spilling is internalized

• Robust at the expense of optimality
  • problem decomposition

• But: instruction selection not yet there!
Instruction Selection

[Based on slides from Gabriel Hjort Blindell]
Graph-based Instruction Selection

int f(int a) {
    int b = a * 2;
    int c = a * 4;
    return b + c;
}

- Represent program as graph

program graph
Graph-based Instruction Selection

- Represent program as graph
- Represent instructions as graph
Graph-based Instruction Selection

 Represent program as graph
 Represent instructions as graph
 Select matches such that program graph is covered

```c
int f(int a) {
    int b = a * 2;
    int c = a * 4;
    return b + c;
}
```
Graph-based Instruction Selection

int f(int a) {
    int b = a * 2;
    int c = a * 4;
    return b + c;
}

- Represent program as graph
- Represent instructions as graph
- Select matches such that program graph is covered
State of the Art

- Local instruction selection
- Program graphs per block
- Graphs restricted to data flow
  - cannot handle control flow such as branching instructions
- Greedy heuristics
  - For example, maximal munch
Instruction Examples

• satadd

• Exists in many DSPs
• Incorporates control flow
• Extends across basic blocks

```c
i = 0
if i < N
  t1 = i * 4
  t2 = A + t1; t3 = B + t1
  a = load t2; b = load t3
  c = a + b
  if MAX < c
    t4 = C + t1
    store t4, c
    i = i + 1
```
Instruction Examples

• satadd
• repeat

• Exists in many processors
  • for example Intel’s x86
• Incorporates control flow
• Extends across basic blocks

Code:

```c
i = 0
if (i < N) {
    t1 = i * 4;
    t2 = A + t1;  t3 = B + t1;
    a = load t2;  b = load t3;
    c = a + b;
    if (MAX < c) {
        t4 = C + t1;
        store t4, c;
        c = MAX;
    } else {
        i = i + 1;
    }
}
```
Instruction Examples

- satadd
- repeat
- add4

- SIMD-style instruction
  - very common
- Requires **global code motion**
  - move computations across blocks
- Depending on hardware may require copying
  - different register file

```plaintext
i = 0
if i < N
  t1 = i * 4
  t2 = A + t1; t3 = B + t1
  a = load t2; b = load t3
  c = a + b
  if MAX < c
    t4 = C + t1
    store t4, c
    i = i + 1
  else
    c = MAX
```
Universal Instruction Selection

- Global instruction selection
- Program graphs for entire functions
- Instruction graphs capture both data and control flow
  - handles broad range of instructions found in today’s processors
- Integrates global code motion
- Takes data-copying overhead into account

- Presupposes an expressive approach such as CP
Program Graph (Example)
Instruction Graph (satadd)
Approach

- Before: create instruction graphs
- Code generation
  - create program graph
  - compute possible matches (standard algorithm VF2 [Cordella ea, 2004])
  - generate model in MiniZinc
  - solve model with CPX 1.0.2
Model Summary

• Decision variables
  • which match is selected?
  • in which block are selected matches placed?
  • in which block is data made available?

• Constraints (selection)
  • operations must be covered by exactly one match
  • control flow cannot be moved
  • data must be defined before used
  • definition edges must be enforced
  • blocks must be ordered (respect fall-through branching if possible)
  • implied and dominance constraints

• Objective functions
  • minimize estimated execution time
  • minimize code size
Experiments

• Benchmarks
  • 16 functions from MediaBench
  • program graphs have 34-203 nodes
  • all models solved to optimality with CPX 1.0.2

• For Simple MIPS32
  • simple RISC architecture: worst-case scenario
  • surprise: 1.4% mean speedup over LLVM 3.4
  • better: global code motion; worse: constant reloading
  • runtimes: 0.3-83.2 seconds, median 10.5 seconds

• For Funky MIPS32 (made up)
  • MIPS32 + common SIMD instructions: good case
  • 3% mean speedup over Simple MIPS32
  • surprise: sometimes SIMD-style is not really that good!
  • runtimes: 0.3-146.8% seconds, median 10.5 seconds
Discussion

• Overcomes many restrictions of state-of-the-art approaches
  • control flow
  • global code motion
  • sophisticated instructions

• Model and representation designed together
  • expressive representation requires expressive models

• Limitations
  • constant reloading
  • if-conversion (predication), well: no approach can do this anyway!
SUMMARY
The Only Important Slide

• Are you interested in combinatorial optimization for compilation?

• Do you want to do a postdoc in one of the most beautiful and dynamic cities in the world?

• Then talk to me!

• Open position at KTH for one year, might be prolonged to two years
  • salary and benefits are good
  • deadline is end of October
Now and Then...

- **Status**
  - instruction scheduling: local, standard
  - register allocation: global, unique
  - instruction selection: global, unique
  - not fully integrated
  - solving pretty naïve

- **Future**
  - instruction scheduling: superblocks, if-conversion (predication)
  - register allocation: rematerialization
  - more sophisticated solving
  - integration!!!
Project & Goals

• Unison has a considerable engineering part
  • processor descriptions (separate large project)
  • robust and maintainable tool chain
  • testing and transfer

• A production-quality tool that will be deployed
  • industrial strength re-implementation started

• An open-source contribution to LLVM
  • legal process started, but need to convince LLVM developers...

• Real significance
  
  simplicity even for today’s freak processors