

Optimal Register Allocation and Instruction Scheduling for LLVM

Roberto Castañeda Lozano – SICS, KTH

joint work with:

G. Hjort Blindell – KTH, SICS

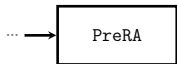
M. Carlsson – SICS

F. Drejhammar – SICS

C. Schulte – KTH, SICS



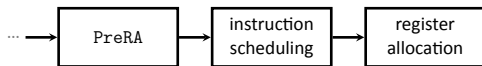
Code Generation in LLVM



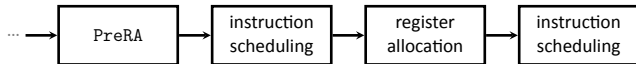
Code Generation in LLVM



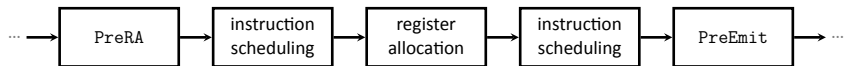
Code Generation in LLVM



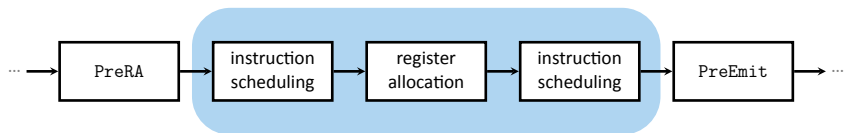
Code Generation in LLVM



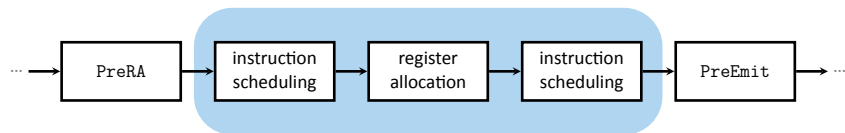
Code Generation in LLVM



Code Generation in LLVM



Code Generation in LLVM



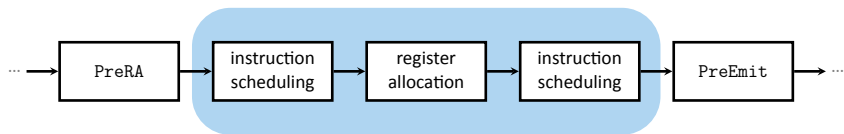
- Stages, heuristics

Code Generation in LLVM



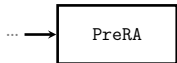
- Stages, heuristics
- Pros: compilation speed

Code Generation in LLVM

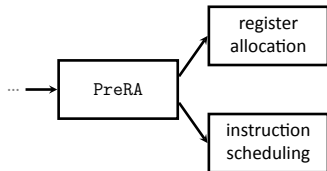


- Stages, heuristics
- Pros: compilation speed
- Cons: suboptimal, complex

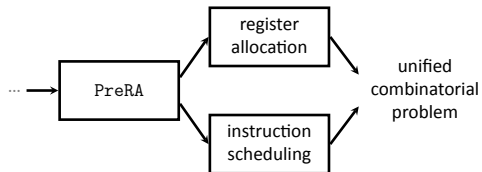
Introducing Unison



Introducing Unison



Introducing Unison

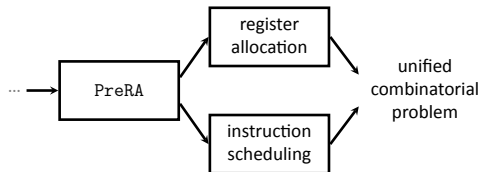


Introducing Unison

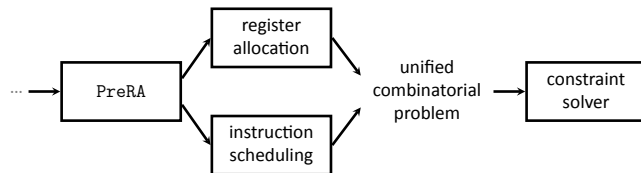


$$\begin{aligned}
 & \text{minimize } \sum_{b \in B} \text{weight}(b) \times \text{cost}(b) \quad \text{subject to} \\
 & l_t \iff \exists p \in P : (\text{use}(p) \wedge y_p = t) \quad \forall t \in T \\
 & a_{\text{definer}(t)} \iff l_t \quad \forall t \in T \\
 & a_o \iff y_p \neq \perp \quad \forall o \in O, \forall p \in \text{operands}(o) \\
 & a_o \iff i_o \neq \perp \quad \forall o \in O \\
 & r_{y_p} \in \text{class}(i_o, p) \quad \forall o \in O, \forall p \in \text{operands}(o) \\
 & \text{disjoint2}(\{\langle r_t, r_t + \text{width}(t) \rangle \times \langle l_t, l_t, l_e_t \rangle : t \in T(b)\}) \quad \forall b \in B \\
 & r_{y_p} = r \quad \forall p \in P : p \triangleright r \\
 & r_{y_p} = r_{y_q} \quad \forall p, q \in P : p \equiv q \\
 & l_t \implies l_t = c_{\text{definer}(t)} \quad \forall t \in T \\
 & l_t \implies l_e_t = \max_{o \in \text{users}(t)} c_o \quad \forall t \in T \\
 & a_o \implies c_o \geq c_{\text{definer}(y_p)} + \text{lat}(i_{\text{definer}(y_p)}) \quad \forall o \in O, \forall p \in \text{operands}(o) : \text{use}(p) \\
 & \text{cumulative}(\{\langle c_o, \text{con}(i_o, r), \text{dur}(i_o, r) \rangle : o \in O(b)\}, \text{cap}(r)) \quad \forall b \in B, \forall r \in R
 \end{aligned}$$

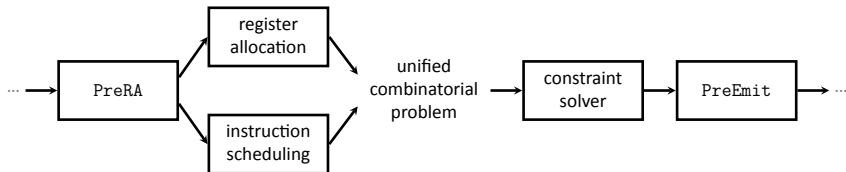
Introducing Unison



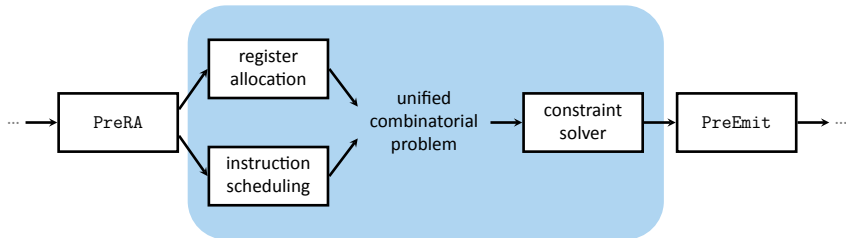
Introducing Unison



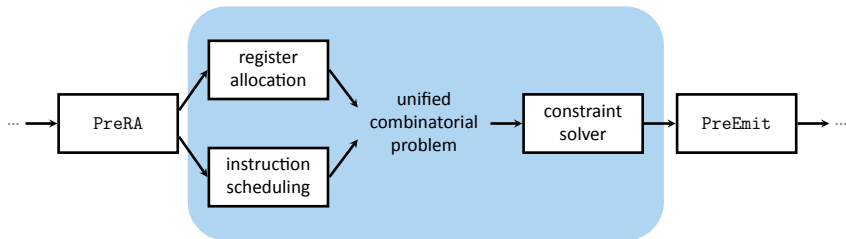
Introducing Unison



Introducing Unison

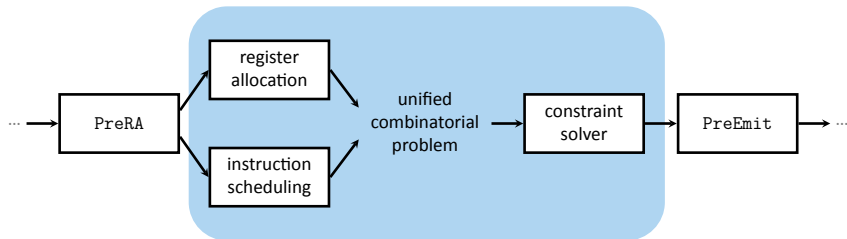


Introducing Unison



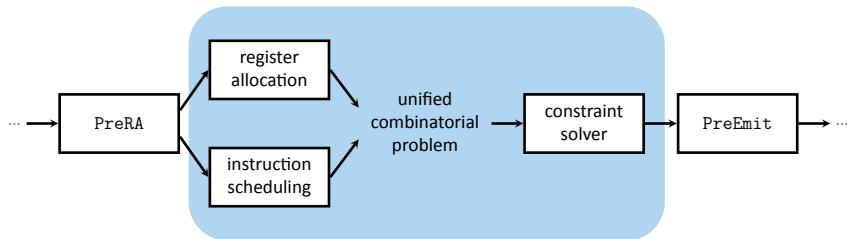
- Integration, combinatorial optimization

Introducing Unison



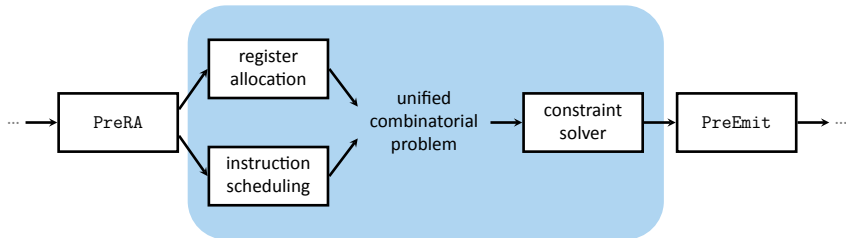
- Integration, combinatorial optimization
- Pros: simple, optimal

Introducing Unison



- Integration, combinatorial optimization
- Pros: simple, optimal
- Cons: compilation slowdown

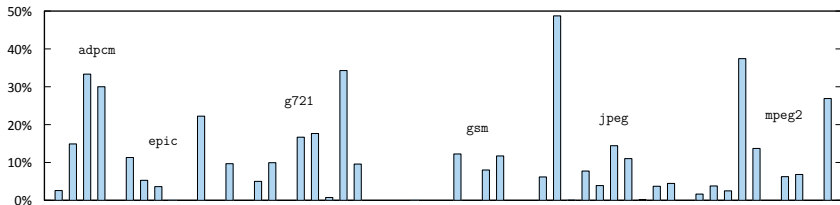
Introducing Unison



- Integration, combinatorial optimization
- Pros: simple, optimal
- Cons: compilation slow

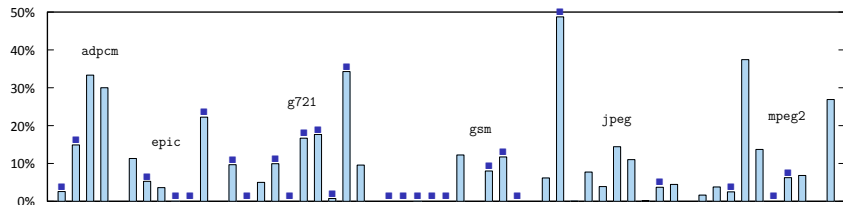
perfect complement
to LLVM!

Speedup over LLVM 3.8



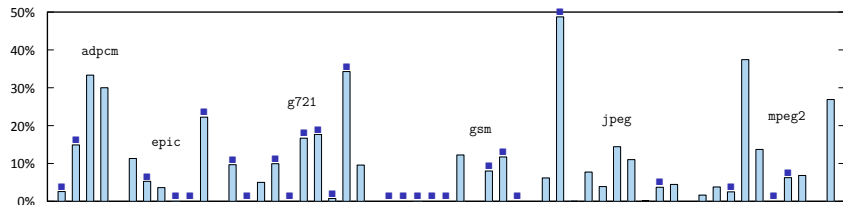
- 50 MediaBench functions
- Hexagon V4 processor

Speedup over LLVM 3.8



- 50 MediaBench functions
- Hexagon V4 processor
- Provably optimal (■) for 54% of the functions

Speedup over LLVM 3.8



- 50 MediaBench functions
- Hexagon V4 processor
- Provably optimal (■) for 54% of the functions
- Compilation time: from seconds to minutes

Unison Is Practical and Effective

- Integrated
 - register allocation
 - instruction scheduling

Unison Is Practical and Effective

- Integrated
 - register allocation
 - instruction scheduling
- Simple, optimal, slower

Unison Is Practical and Effective

- Integrated
 - register allocation
 - instruction scheduling
- Simple, optimal, slower
- Complements LLVM:
 - traditional LLVM for compile/debug cycle

Unison Is Practical and Effective

- Integrated
 - register allocation
 - instruction scheduling
- Simple, optimal, slower
- Complements LLVM:
 - traditional LLVM for compile/debug cycle
 - LLVM + Unison for release builds

Unison Is Practical and Effective

- Integrated
 - register allocation
 - instruction scheduling
- Simple, optimal, slower
- Complements LLVM:
 - traditional LLVM for compile/debug cycle
 - LLVM + Unison for release builds
- Useful analysis tool for LLVM developers
 - how good is my heuristic?

Demo at CC2016 (12:20)

The screenshot displays the Factorial: Unison vs. LLVM solver interface. The main window shows the source code for a factorial function with labels like `.LBB0_0`, `.LBB0_1`, and `.LBB0_2`. A control flow graph (CFG) is visible in the center, showing nodes and edges with instructions such as `l[011] = stw`, `l[012] = stw`, and `l[013] = ldw`. A register array window is open, showing a grid of registers (R0-R3, P0-P1, m0-m3) across cycles (0, 1, 2). A data flow graph is also visible on the right, showing nodes and edges with instructions like `r[011] = R0` and `r[012] = R0`. The bottom status bar shows the current state: `[factorial with alterna... solver (b1) factorial: Unison vs. LL... Register array Data flow`.

www.sics.se/~rcas/unison-demo