Autovectorization in LLVM

Christopher Czyba, Hermann Walth

July 21, 2014

Overview

1. Motivation
2. Overview of LLVM
3. Loop-vectorization in LLVM
   - Basic Flow of Loop Vectorization
   - Loop Legalizer - Requirements and Examples

Motivation - SSE and AVX - A short history overview

SIMD Streaming Extension and Advanced Vector eXtension are SIMD extensions for the x86/x86_64 architecture.

Overview over the history

- MMX - 64 bit MM-registers, for 64,32,16 and 8 bit integers
- SSE - Adds 128 bit XMM-registers for floats
- SSE2 - XMM registers may be used for integers and doubles
- SSE3/4 - More operations like max/min, mostly for graphics
- AVX - Adds 256 bit YMM-registers, only for floats and doubles

Analogous instructions exist for ARM, PPC and other architectures. GPUs are based on the SIMD Model.

LLVM Compilation Flow

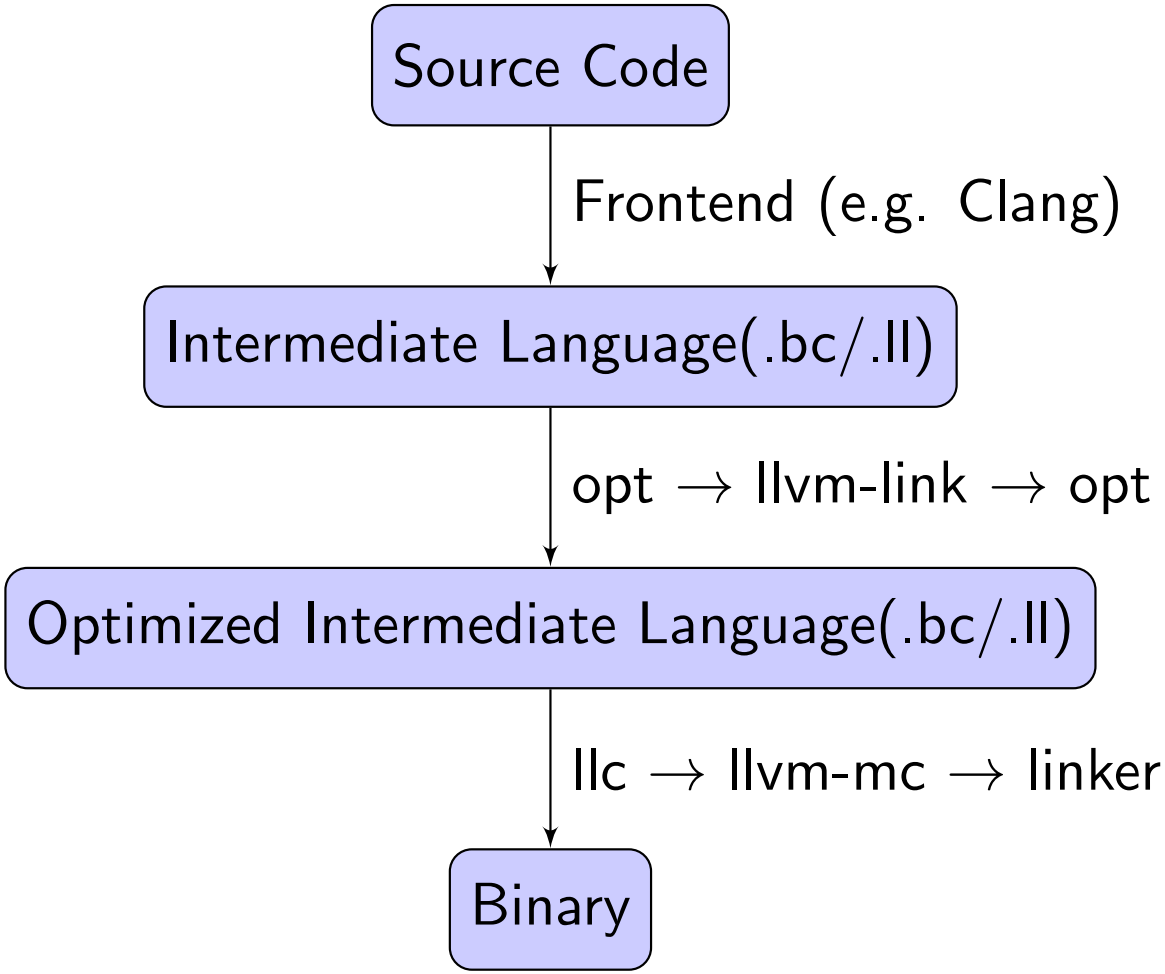


Figure: Basic LLVM Compilation Flow

LLVM Compilation Flow

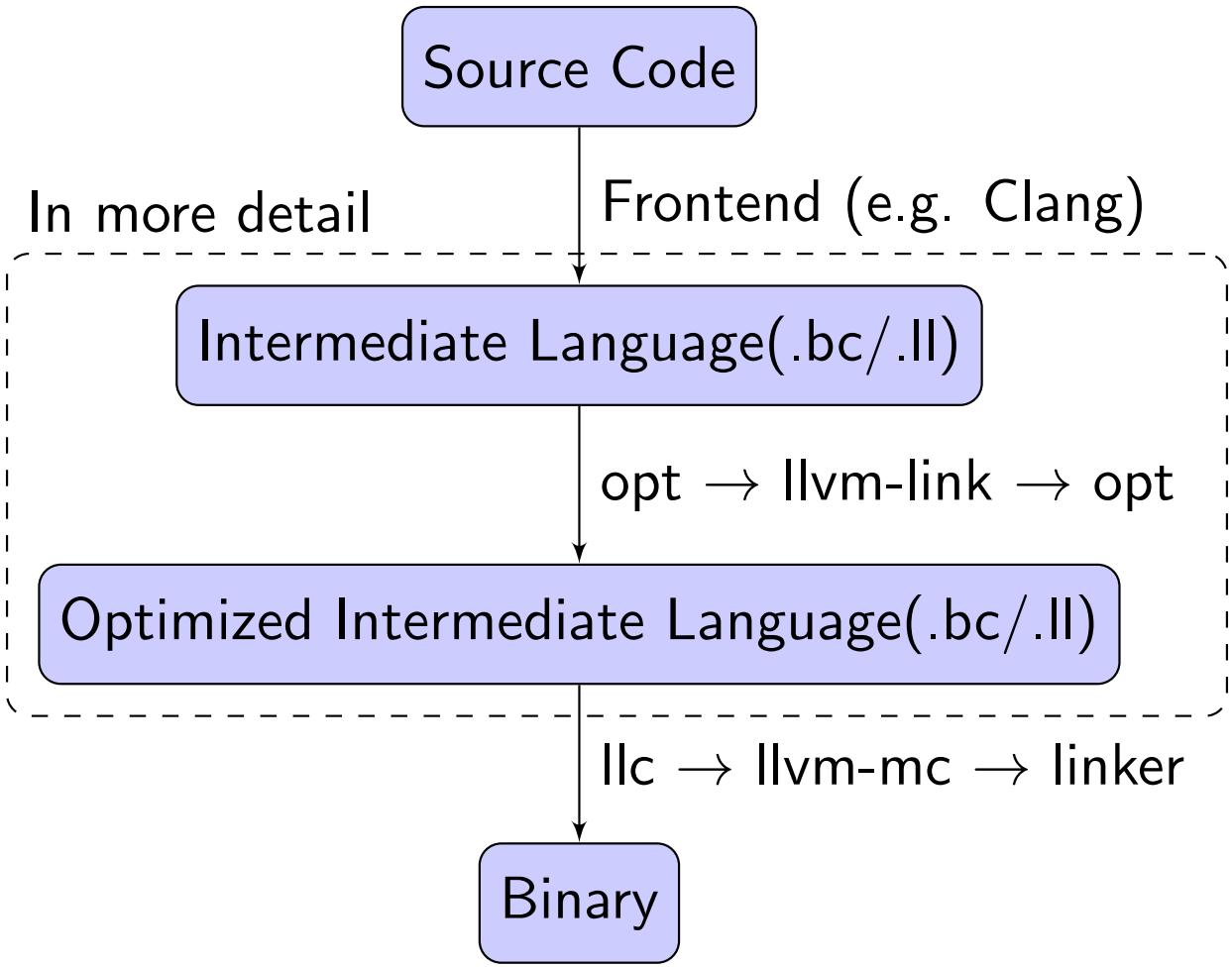


Figure: Basic LLVM Compilation Flow

Overview of opt

- opt = collection of analyses and transformations, e.g.
  - Dead Store/Global/Argument/Code Elimination
  - Constant Calculation and Replacement
  - Loop Vectorization

Overview of opt

- opt = collection of analyses and transformations, e.g.
  - Dead Store/Global/Argument/Code Elimination
  - Constant Calculation and Replacement
  - Loop Vectorization
- -Ox option selects a (more or less) sane order of those
  - Often reruns analyses and transformations multiple times
  - Can be printed with -debug-pass=Arguments

Overview of opt

- opt = collection of analyses and transformations, e.g.
  - Dead Store/Global/Argument/Code Elimination
  - Constant Calculation and Replacement
  - Loop Vectorization
- -Ox option selects a (more or less) sane order of those
  - Often reruns analyses and transformations multiple times
  - Can be printed with -debug-pass=Arguments
- Many have (implicit) prerequisites (i.e. loop vectorizer)

LLVM Intermediate Language (IL)

- Primitive Types `float i1 i32 i8*`
- Vector Types `<N x T>`
  for vector length `N`, primitive type `T`
- Instructions that work on primitive types

  ```
  %x = add i32 %a, %b
  ```

  also work on vectors

  ```
  %y = add <4 x i32> %v1, %v2
  ```

- Vector values are mapped to SIMD registers on target machine

Goals Loop Vectorization

Goals:

- Unroll loops for less overhead
- Use SIMD instructions to process multiple data at once

$$\begin{array}{c} a_1 := b_1 \odot c_1 \\ \vdots \\ a_n := b_n \odot c_n \end{array} \Rightarrow \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} := \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \odot \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

SISD SIMD

- Sometimes usage of multiple processors
- Linearizing Control Flow (Also called flattening)

Basic flow of Loop Vectorization in LLVM

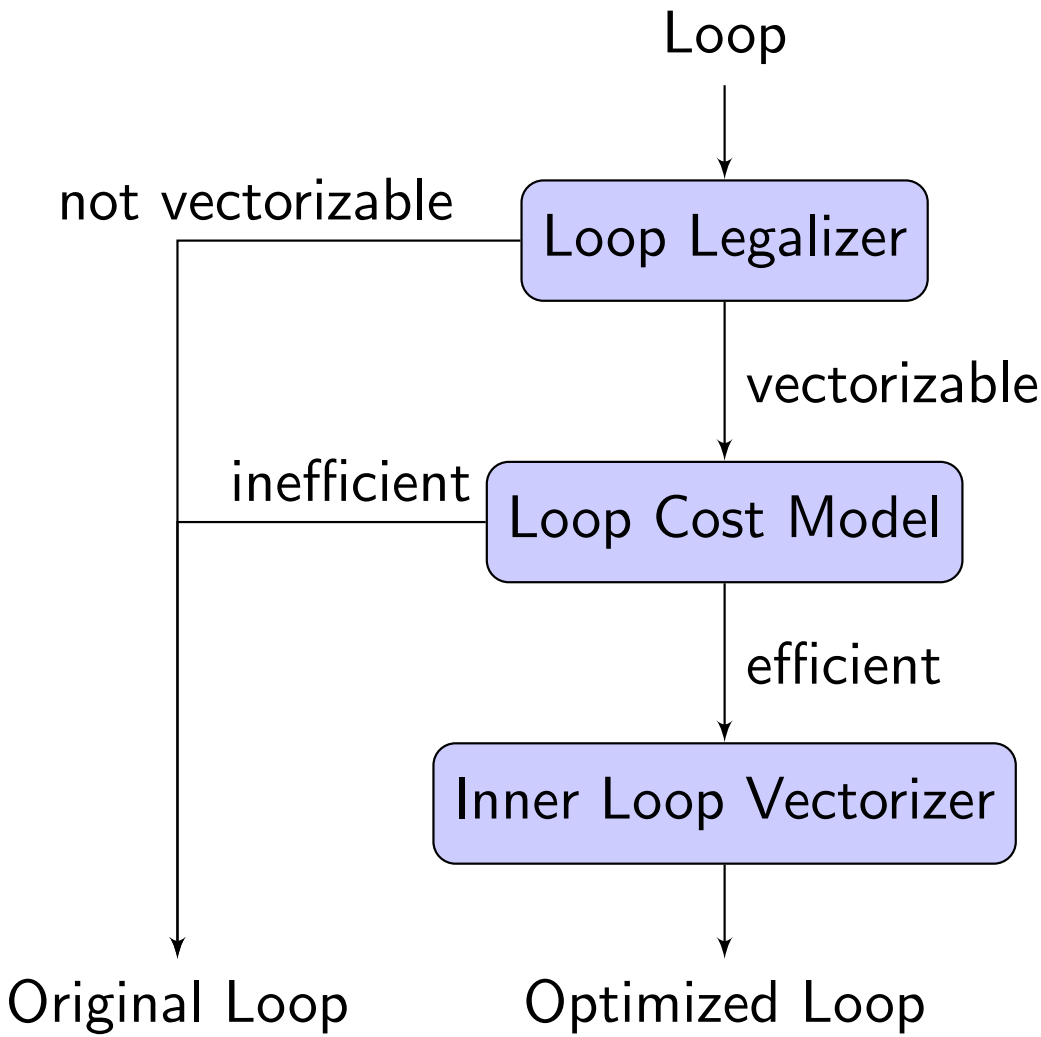


Figure: Basic Loop Vectorization Flow

Structural Vectorization of Loops

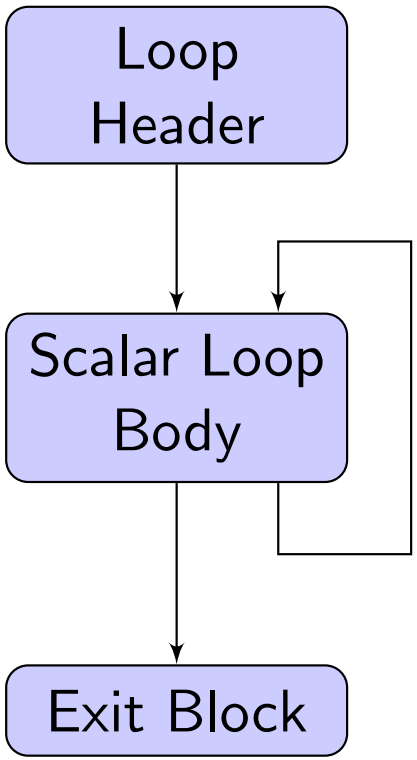


Figure: Original Loop

Structural Vectorization of Loops

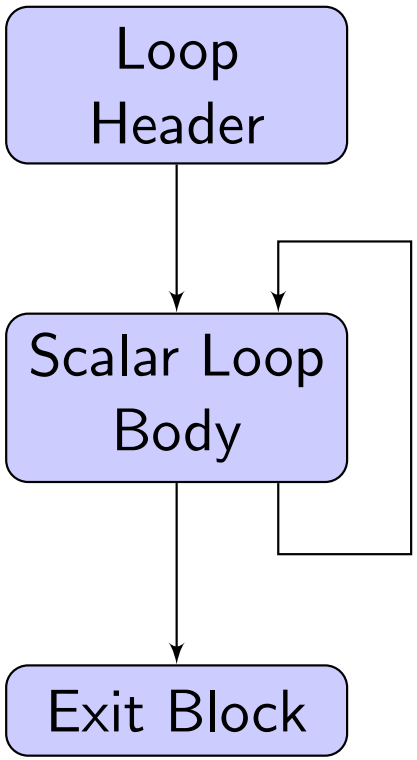


Figure: Original Loop

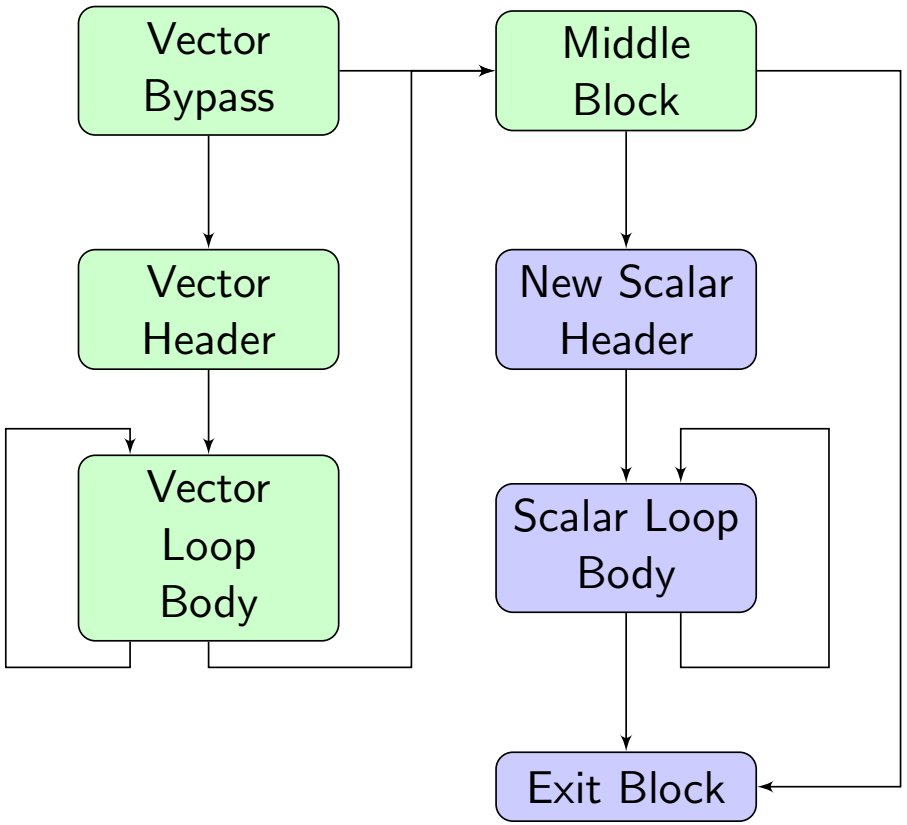


Figure: Vectorized Loop

Jobs of the Legalizer

Obvious:

- Checks whether it is legal to vectorize the loop or not

Jobs of the Legalizer

Obvious:

- Checks whether it is legal to vectorize the loop or not

Non-obvious:

- Chooses vectorization factor (depends on SIMD instructions)
- Identifies
  - Reduction Variables
  - Induction Variables

Jobs of the Legalizer

Obvious:

- Checks whether it is legal to vectorize the loop or not

Non-obvious:

- Chooses vectorization factor (depends on SIMD instructions)
- Identifies
  - Reduction Variables
  - Induction Variables

In short: Determines *how* to vectorize the loop

Requirements for applying loop-vectorization

- Only vectorizes innermost loop (innermost-loop.c)
- Strides of exactly 1 (non-one-strides.c)
- IL needs to be optimized beforehand (needs-preoptimization.c)
- Division within the loop (may trap, division-in-loop.c)
- C++-class induction/reduction fails (IntegerWrapper.cpp)

Requirement - Single Backwards edge

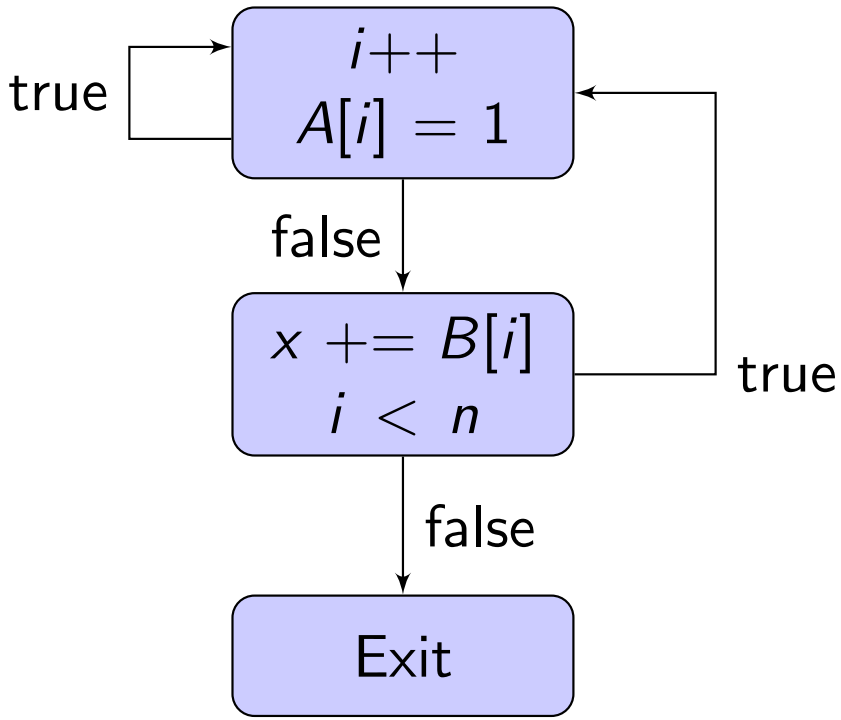


Figure: Two Backward Edges, Loop of multiple-backward-edges.c

Requirement - Single Unique Exit

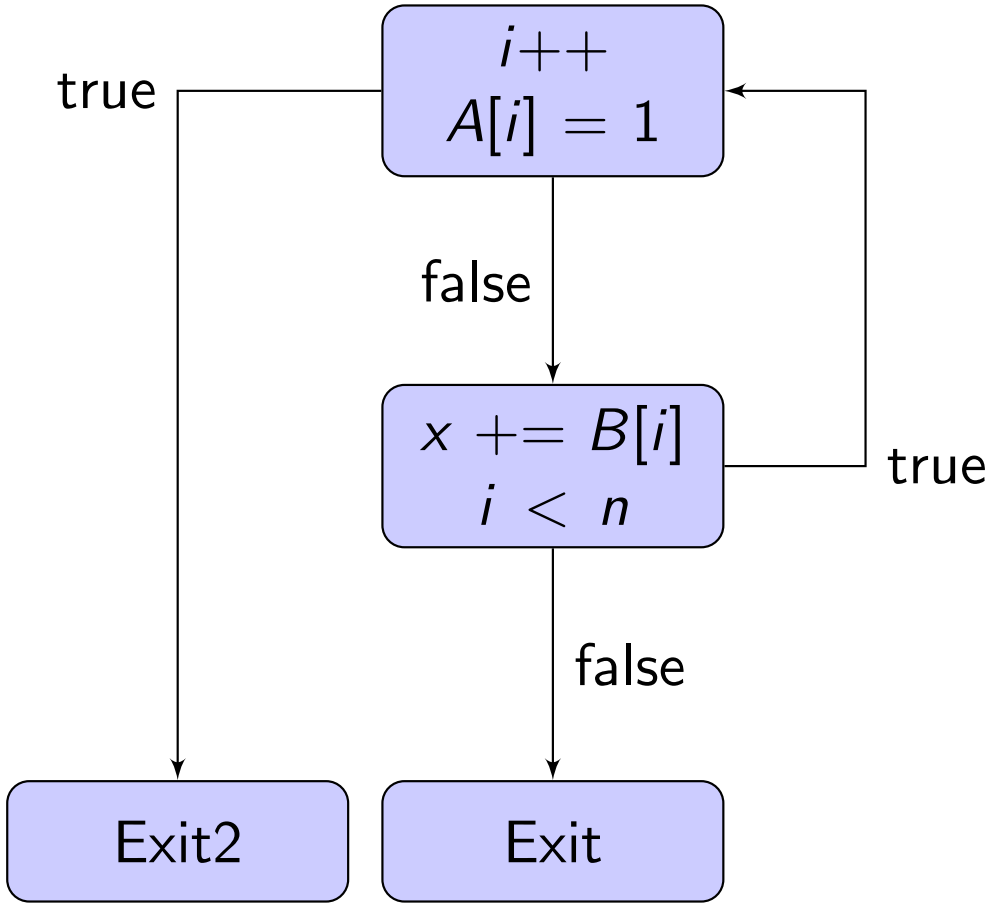
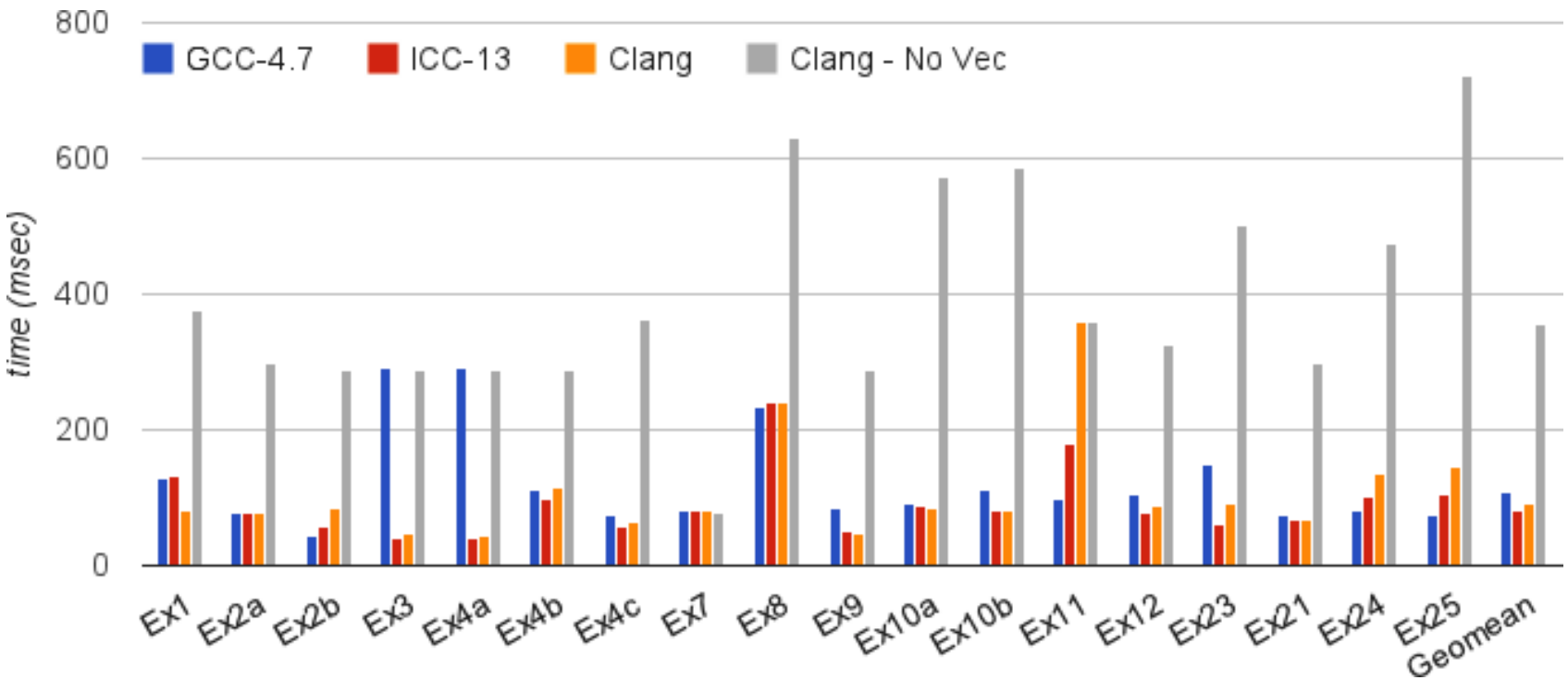


Figure: Two distinct exit nodes, Loop of different-exit-blocks.c

Conclusion

Despite all this, LLVM is still a good choice for Loop Vectorization:

Figure: Comparison of vectorization performance for GCC, Clang and Intel C Compiler



Sources

- LLVM Source: `LoopVectorize.cpp` - See `http://llvm.org/doxygen/LoopVectorize_8cpp_source.html`
- `http://llvm.org/docs/Vectorizers.html`
- Code examples from `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`