Sign Extension Optimizations inside LLVM

Panagiotis Karouzakis, Polyvios Pratikakis

University Of Crete, ICS-FORTH

11/4/2024
Extension Operations

Sign extend the low 4 bits into a 8 bit value.

Zero extend the low 4 bits into a 8 bit value.

Truncate the 8 bit value into a 4 bit value. The high bits are lost.
The Problem

Figure 1: X86 DAG code after the last DAG combine

We could drop the truncate and the any_extend and do a i32 addition. The truncate does not generate any code in X86 but the any_extend to i32 does.
In C/C++ code the type \textit{int} may create many redundant sign extensions.

Sign extension optimizations may need further research especially for RISC-V

On X86 redundant truncations followed by any\_extend are observed on SPEC 2017 benchmarks in many blocks.

LLVM still doesn’t query for all the legal widths that a operator has for a given target. It does that on a subset of the available operators.
Kevin Redwine and Norman Ramsey [CC 2004] proposed the \textit{filltypes}.

They created type rules that used the \textit{filltypes} to find the optimal solution.

They used Dynamic Programming to find a solution.

Their implementation was on the small Language C—a subset of C.

They visited each AST node to apply all the type rules.
We use the notion of filltype proposed by Kevin Redwine et al[CC 2004 ]

A filltype indicates what an operand produces and accepts in their upper bits.
  - Upper bits are the rest of the bits that do not have any data.

An operand has a fill type only if it is widenable, i.e., if applying the operator to wide values can simulate the operator applied to narrow values.

For example, since xor has a filltype, xor\textsubscript{i32} can be implemented as xor\textsubscript{i64} regardless of the high bits of the operands.

\[
(\overline{1010}) \oplus (\overline{0011}) = 1001
\]
\[
(\overline{11111010}) \oplus (\overline{11110011}) = 00001001
\]

4 bit xor implemented using 8 bit xor
We use symbols $s$, $z$, $g$ to match sign, zero, and garbage upper bits respectively.

The input operands must be typed before visiting an instruction and they can be an instruction as well.

\[
\begin{align*}
\text{and} & :: g \times g \rightarrow g \\
\text{and} & :: z \times g \rightarrow z \\
\text{and} & :: g \times z \rightarrow z \\
\text{and} & :: s \times s \rightarrow s
\end{align*}
\]
Legal Operations

- Create multiple solutions per Instruction that are legal for the target with different Instruction width.
- Use data flow information to learn how many bits are data.
- If we have upper bits left and not all of them are data we have a filltype.
- If not we can insert an extension to add a filltype based on the target.
- If we insert a truncation we can potentially remove a filltype.
Binary operators

- For binary operators search for legal filltype rules.
  - $xor :: s \times s \rightarrow s$
  - $xor :: z \times z \rightarrow z$
  - $xor :: g \times g \rightarrow g$
  - ...

- Ask Target Lowering for the legal Instructions widths.
- For example, many targets offer xor with 32 bits and 64 bits.
  - $xor :: 32 \times 32 \rightarrow 32$
  - $xor :: 64 \times 64 \rightarrow 64$

- If we have found an operation with legal Instruction width that has a fillType, we can create a solution that keeps the new width, the data bits and other information.
Our approach

- Doing this optimization inside LLVM IR to add support for every Language Frontend.
- As a consequence we solve a flow sensitive problem to deal with the control flow, instead of a flow insensitive problem.
- We extend the proposed operand *filltypes* to match the LLVM operands.
- We have to deal with LLVM Intrinsics i.e., to choose among Intrinsics of different widths.
- While we use the available target operations, X86 needs special handling because some extensions and truncations are free.
- When we have more than 1 Use of an Instruction we might get conflicting solutions.
Multiple Users Example

Instruction

Solutions

User1

User2
PHIs example

```
.. → phi
  ↓     ↓
sub    add
  ↓     ↓
sext   xor
  ↓     ↓
sub    div
```

Karouzakis et al. (UOC, FORTH)  LLVM Sign Extensions  11/4/2024 12 / 15
Implementation details

- Iterate all the Integer Instructions
- Solve Instruction

No

Dataflow Fixpoint

Yes

Finish Solving Instructions
- Search best Solution using DP
- Apply the best Solution
Limitations and Next Steps

- If an operator overflows it requires special consideration.
- Checking for overflows is not one hundred percent accurate, so we lose optimization opportunities.
- Currently the project is implemented as a Function Pass. It may be useful to use Module pass to better infer function parameters.
- Using preferred X86 register width.
- Choosing between Intrinsics of different widths.
Thank You!