Silent stores optimization in LLVM: overview and installation guide
(version 1.0)

August 2017

Authors: Rabab Bouziane, Erven Rohou and Abdoulaye Gamatie

Contents

1

About this guide

This document helps the reader to get an overview about the silent stores optimization, and to install an LLVM-based implementation. It is developed within the French ANR CONTINUUM project [1]. We briefly describe the optimization, then we present the associated compilation framework.

[1] http://www.lirmm.fr/continuum-project

2

General presentation of the optimization

2.1 Introduction

The silent stores optimization, called later *SS*, is an optimization built in LLVM 3.8 (https://releases.llvm.org/3.8.0/docs/ReleaseNotes.html), see Figure 2.1.

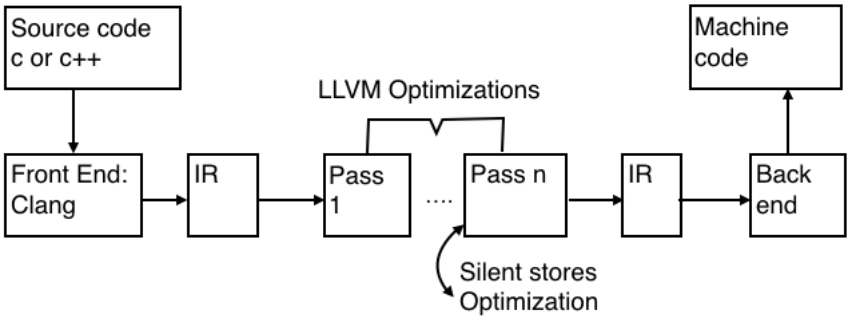

Figure 2.1: LLVM architecture

This transformation aims to eliminate *silent stores* from a program. A store is said silent if it writes a value to a memory address where the same value is already stored. The *SS* optimization acts on the Intermediate Representation (IR) and consists of two steps:

- a profiling step that detects the silent stores, and
- their positions in the code and a transformation step that transforms the silent stores in order to avoid their execution.

The above *SS* optimization has been studied in the context of the CONTINUUM ANR project. The objective is to reduce the energy consumption in architectures based on non volatile memories such as STT-RAM by reducing the number of stores. Indeed, such memory technologies are currently known to have penalizing write (i.e. store) operations in terms of latency and power consumption.

The silent stores optimization relies on a compilation process consisting of two main steps, described in the sequel.

2.2 Profiling step

First of all, the profiling step is executed and through a value locality analysis, a number of store's characteristics are defined: the number of instances, the number of silent

instances and their positions in the code. Concretely, during the compilation of a program, new functions are called whenever a store is encountered. These functions check if the store is silent and are associated to two counters; the first one counts the number of instances and the second one counts the number of silent instances. We do so to be able to determine the highly silent stores and the slightly silent stores. The output of this step is a profiling file, see Figure 2.2, that is used for the second step of the compilation framework to perform the transformations.

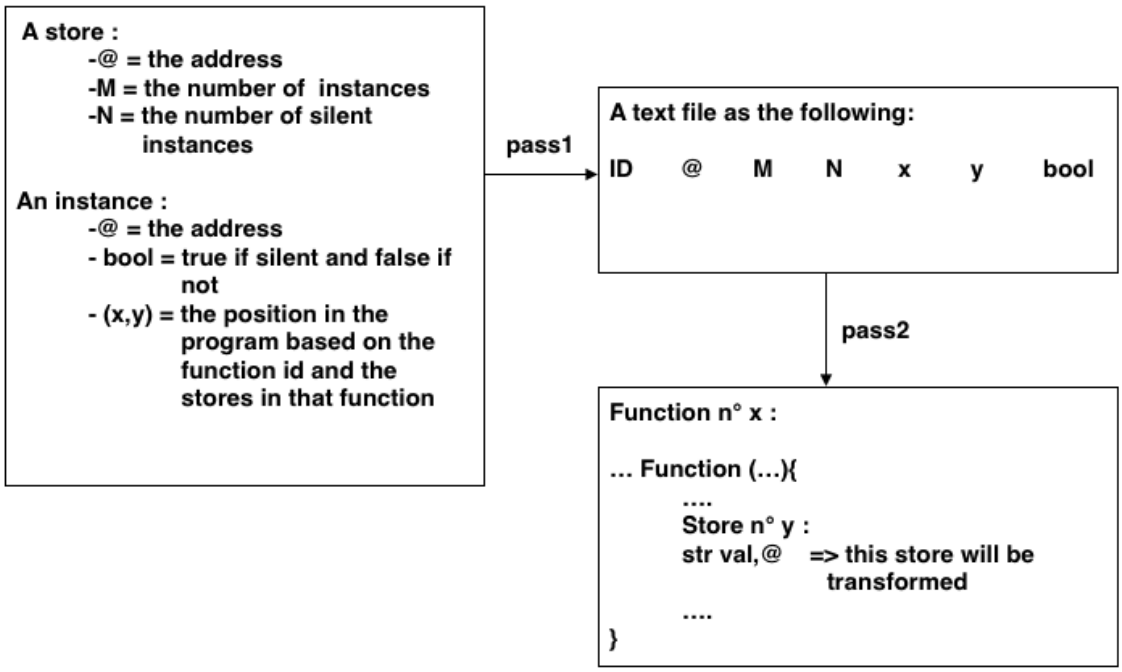

Figure 2.2: Design of the compilation framework

2.3 Transformation step

The transformation consists in transforming a store known as silent into a set of instructions: a load instruction at the store address, a comparison instruction, to compare the written value to the already stored value and a conditional branch instruction to skip the store if needed (see pseudo code on Figure 2.3). These newly inserted instructions allow to verify that the detected silent store is indeed silent.Therefore, with the profiling, the verification process is not performed for all the stores, reducing hence the overhead of the transformation. Please refer to the user guide to see how the compilation framework works in details.

```
1    store @x = val
```

(a) original

```
1    load y = @val
2    cmp val, y
3    bEQ next
4    store @x, val
5  next:
```

(b) transformed

Figure 2.3: Silent store elimination: original code stores `val` at address of `x`; transformed code first loads the value at address of `x` and compares it with the value to be written, if equal, the branch instruction skips the store execution.

3

How to install the optimization framework

The silent stores optimization implementation in LLVM can be retrieved from the corresponding source code directory available in the following link:

https://gitlab.inria.fr/rbouzian/Silent_stores_pass

Then, the corresponding LLVM transformation pass is called **Silent_stores_pass** and can be installed by the user in his/her local directory as follows:

```
$ cd Silent_stores_pass
$ mkdir build
$ cd build
$ cmake ..
$ make
$ cd ..
```

To use the optimization framework once installed, the user can refer to the instructions indicated in the separate User Guide document (on the CONTINUUM project web page).