LLVM PGO Instrumentation: Example of CallSite-Aware Profiling

Authors:
Pavel Kosov, Sergey Yakushkin
Agenda

• Target of this presentation
• PGO Introduction
• Implementation details of current PGO
• Implementation details of proposed extension
• Description of llvm-profdata
• How compiler load and use profdata in optimizations.
Target of this presentation

• Make a general description of PGO
  • What is PGO?
  • When it is useful and when its not?
  • Case study

• Describe how PGO works

• Describe how to make an extension for PGO
PGO Introduction

PGO not an optimization but an approach

**Pros**: can improve important scenario(s)
**Cons**: other(s) scenarios may degrade

Profile data – describes a scenario of program usage

Possible ways for generation of profile data:
- Sampling
- Instrumentation
PGO Introduction. Sampling Pipeline

1. Build the code with source line table information
   > clang++ -O2 -gline-tables-only code.cc -o code

2. Run the executable under a sampling profiler
   > perf record -b ./code

3. Convert the collected profile data to LLVM format
   > create_llvm_prof --binary=./code --out=code.prof

4. Build the code again using collected profile
   > clang++ -O2 -gline-tables-only \
     -fprofile-sample-use=code.prof code.cc -o code
PGO Introduction. Sampling Formats

- **ASCII** text – profile info per section
- **Binary** encoding: compact format produced by autofdo create_llvm_prof tool
- **GCC** encoding: gcov compatible encoding, produced by autofdo create_gcov tool
PGO Introduction. Instrumentation Pipeline

1. Build an instrumented version of the code
   > clang++ -02 -fprofile-instr-generate code.cc -o code
2. Run the instrumented executable with necessary inputs
   > LLVM_PROFILE_FILE="code-%p.profraw" ./code
3. Combine profiles from multiple runs and convert the “raw” profile format to the input expected by clang
   > llvm-profdata merge -output=code.profdata code-*.profraw
4. Build the code again using collected profile data
   > clang++ -02 -fprofile-instr-use=code.profdata code.cc \
      -o code
PGO Introduction. Kinds of Instrumentation

- Front-end (FE) - `fprofile-instr-generate`
- Middle-end (IR) - `fprofile-generate`
- Middle-end context sensitive (IR CS) - `fcs-profile-generate`
PGO Introduction. IR CS Instrumentation

> clang++ -02 -fprofile-use=code.profdata \ -fcs-profile-generate -o cs_code

> ./cs_code

> llvm-profdata merge -output=cs_code.profdata code.profdata

> clang++ -02 -fprofile-use=cs_code.profdata
Implementation details (current state)

- All about instrumentation
- MST-based insertion of counters
- Insertion of value probes
- Lowering of intrinsics
- Compiler-rt built-in functions for PGO
  - Values updating
  - Open file, merge values and counters
- Example
- Format of profdata in RAM and on disc
PGO Introduction. Instrumentation Pipeline

PGO implementation steps

- Intrinsics insertion
- Intrinsics lowering

Instrumentation

- Write / merge profraw data
- Convert / merge profdata

Runtime and postprocessing

- Load profdata
- Store in internal structures
- Use in optimizations

Optimization

[Diagram showing the flow of steps with connections between them]
Intrinsics and its lowering

• Intrinsic is a built-in function which is inserted by compiler
• It is used to optimize:
  • Memory calls
  • Floating point operations (fadd/fmul/sin/cos …)
  • PGO counters insertion in our case
  • And more (details: include/llvm/IR/Intrinsics.td)
• When compiler meets intrinsic, it performs “lowering” – replace intrinsic with code or calls to optimized versions of some library functions
MST-based insertion of counters

CFG

entry

if.then

if.then2

exit

if.else

if.else2
MST-based insertion of counters

CFG

entry

if.then

if.then2

exit

if.else

if.else2

MST

entry

if.then

if.then2

exit

if.else

if.else2
MST-based insertion of counters
Value probes

- Record indirect calls
- Record sizes for memcpy / memmove / memset

- If target value is in green area, then it is set to PreciseRangeLast + 1
- If target value is in blue area, then it is set to LargeValue
- Otherwise it is recorded with its real value
Compiler-rt builtin functions

- Builtins for profiling
  - __llvm_profile_get_magic
  - __llvm_profile_get_version
  - __llvm_profile_instrument_target
  - __llvm_profile_instrument_memop
  - __llvm_profile_write_file
  - __llvm_profile_dump
  - __llvm_profile_reset_counters
  - and much more (details in compiler-rt/lib/profile)
void foo(char* buf, int num) {
    memset(buf, 0, num);
}

define dso_local void @foo(i8* nocapture %0, i64 %1) {
    %3 = load i64, i64*
    getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo,
    i64 0, i64 0)
    %4 = add i64 %3, 1
    store i64 %4, i64*
    getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo,
    i64 0, i64 0)
    tail call void @__llvm_profile_instrument_memop(i64 %1,
    i8* bitcast ({ i64, i64, i64*, i8*, i8*, i32, [2 x i16] }
    * @__profd_foo to i8*), i32 0)
    tail call void @llvm.memset.p0i8.i64(i8* align 1 %0, i8 0, i64 %1, i1 false)
    ret void
}
Format of profdata

Header:
• Magic / version
• Paddings
• Sizes of all sections
Format of profdata

Header:
- Magic / version
- Paddings
- Sizes of all sections

ProfData:
- FuncHash, FuncNameMD5
- Pointers (offsets) to related data in sections
- Number of counters
**Format of profdata**

**Header:**
- Magic / version
- Paddings
- Sizes of all sections

**ProfDatas:**
- FuncHash, FuncNameMD5
- Pointers (offsets) to related data in sections
- Number of counters
Implementation details for extension

• Overview of proposed extension
  • Example. Pros and cons

• New intrinsics and its lowering

• Change in profdata format

• Extension of internal structures
Overview of Callsite-Aware PGO

• Main difference between original PGO and this one – separate counters for every callsite. It can be useful for several optimizations like Inlining

• Pros: compiler can get more info for enhance optimizations

• Cons: code size, compile time, runtime memory and performance overhead
Overview of callsite-aware PGO

Original PGO

void foo() {
    if (cond) {
        counter1++;
        block1;
    } else {
        counter2++;
        block2;
    }
}

Callsite-aware PGO

void foo() {
    choose_cs_set;
    if (cond) {
        counter1++;
        block1;
    } else {
        counter2++;
        block2;
    }
}
Overview of callsite-aware PGO Implementation

Pipeline

Intrinsics insertion → Intrinsics lowering → Instrumentation

Write / merge profraw data → Convert / merge profdata → Runtime and postprocessing

Load profdata → Store in internal structures → Optimization

Use in optimizations
Overview of callsite-aware PGO Implementation

Pipeline

- **Intrinsics insertion**
- **Intrinsics lowering**
- **Instrumentation**
- **Runtime and postprocessing**
- **Optimization**

- Write / merge profraw data
- Convert / merge profdata
- Load profdata
- Store in internal structures
- Use in optimizations
Implementation of callsite-aware PGO

- Right before every callsite we need to insert intrinsic which will provide pointer to necessary counters
- How to add this intrinsic:

```c
#include/llvm/IR/Intrinsics.td

// A call to provide a pointer to callsite counters
// 1st parameter - callee hash
// 2nd parameter - callsite id
def int_instrprof_callsite_counters :
    Intrinsic<[], // Ret type(s)
        [llvm_i64_ty, llvm_i32_ty], // Parameters
        []>; // Properties
```
Implementation of callsite-aware PGO

- include/llvm/IR/IntrinsicInst.h

    /// This represents the llvm.instrprof_callsite_counters intrinsic.
    class InstrProfCallsiteCounters : public IntrinsicInst
    {
        public:
            static bool classof(const IntrinsicInst *I) {
                return I->getIntrinsicID() == Intrinsic::instrprof_callsite_counters;
            }

            ConstantInt *getCalleeHash() const {
                return cast<ConstantInt>(const_cast<Value * >(getArgOperand(0)));
            }

            ConstantInt *getCallsiteID() const {
                return cast<ConstantInt>(const_cast<Value * >(getArgOperand(1)));
            }
    }
Implementation of callsite-aware PGO

• Insert intrinsic before each call (if callee is registered)

lib/Transforms/Instrumentation/PGOInstrumentation.cpp

```cpp
const int32_t CallsiteID = getCallsiteId(CalleeHash, CallInstr);
If (CallsiteID > 0) {
    Builder.CreateCall(
        Intrinsic::getDeclaration(M, Intrinsic::instrprof_callsite_counters),
        { Builder.getInt64(CalleeHash),
          Builder.getInt32(CallsiteID) });
}
```
Implementation of callsite-aware PGO

• Example (IR Dump After PGOInstrumentationGenPass):

```cpp
define dso_local void @bar() #0 {
  entry:
    call void @llvm.instrprof.increment(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @__profn_bar, i32 0, i32 0), i64 742261418966908927, i32 1, i32 0)
    call void @llvm.instrprof.callsite.counters(i64 6699318081062747564, i32 3)
    call void @foo()
    call void @llvm.instrprof.callsite.counters(i64 6699318081062747564, i32 0)
    ret void
}
```
Implementation of callsite-aware PGO

- Lowering of the intrinsic
lib/Transforms/Instrumentation/InstrProfiling.cpp

```cpp
bool InstrProfiling::lowerIntrinsics(Function *F) {
    ...
    } else if (auto *CSCounters = dyn_cast<InstrProfCallsiteCounters>(Instr)) {
        lowerCSCounters(CSCounters);
        MadeChange = true;
    }
    ...

lowerCSCounters – record callsite id to a memory location (every function has its own)
Implementation of callsite-aware PGO

• Example (IR Dump After Frontend instrumentation-based coverage lowering)

```c
define dso_local void @bar() #0 {
  entry:
  %pgocount = load i64, i64* getelementptr inbounds ([1 x i64],
    [1 x i64]* @__profc_bar, i64 0, i64 0), align 8
  %0 = add i64 %pgocount, 1
  store i64 %0, i64* getelementptr inbounds ([1 x i64],
    [1 x i64]* @__profc_bar, i64 0, i64 0), align 8
  store i32 3, i32* @_llvm_prof_foo_csid, align 4
  call void @foo()
  store i32 0, i32* @_llvm_prof_foo_csid, align 4
  ret void
}
```
Implementation of callsite-aware PGO

- Example (IR Dump After Frontend instrumentation-based coverage lowering)

```c
define dso_local void @foo() #0 {
  entry:
    %load_csid = load i32, i32* @__llvm_prof_foo_csid, align 4
    %0 = mul i32 %load_csid, 1
    %1 = add i32 %0, 0
    %2 = getelementptr inbounds [5 x i64], [5 x i64]* @__profc_foo, i32 0, i32 %1
    %pgocount = load i64, i64* %2, align 8
    %3 = add i64 %pgocount, 1
    store i64 %3, i64* %2, align 8
    ret void
}
```
Implementation of callsite-aware PGO

• Example (IR Dump After Frontend instrumentation-based coverage lowering)

define dso_local void @foo() #0 {
  entry:
    %load_csid = load i32, i32* @__llvm_prof_foo_csid, align 4
    %0 = mul i32 %load_csid, 1
    %1 = add i32 %0, 0
    %2 = getelementptr inbounds [5 x i64], [5 x i64]* @__profcount_foo, i32 0, i32 %1
    %pgocount = load i64, i64* %2, align 8
    %3 = add i64 %pgocount, 1
    store i64 %3, i64* %2, align 8
    ret void
}
Implementation of callsite-aware PGO

• Example (IR Dump After Frontend instrumentation-based coverage lowering)

define dso_local void @foo() #0 {
  entry:
    %load Csid = load i32, i32* @_llvm_prof_foo_csid, align 4
    %0 = mul i32 %load Csid, 1
    %1 = add i32 %0, 0
    %2 = getelementptr inbounds [5 x i64], [5 x i64]* @_prof_foo, i32 0, i32 %1
    %pgocount = load i64, i64* %2, align 8
    %3 = add i64 %pgocount, 1
    store i64 %3, i64* %2, align 8
    ret void
}
Implementation of callsite-aware PGO

- Example (IR Dump After Frontend instrumentation-based coverage lowering)

```assembly
define dso_local void @foo() #0 {
  entry:
  %load_csid = load i32, i32* @__llvm_prof_foo_csid, align 4
  %0 = mul i32 %load_csid, 1
  %1 = add i32 %0, 0
  %2 = getelementptr inbounds [5 x i64], [5 x i64]* @__prof_foo, i32 0, i32 %1
  %pgocount = load i64, i64* %2, align 8
  %3 = add i64 %pgocount, 1
  store i64 %3, i64* %2, align 8
  ret void
}
```
Implementation of callsite-aware PGO

- Example (IR Dump After Frontend instrumentation-based coverage lowering)

```assembly
define dso_local void @foo() #0 {
  entry:
  %load_csid = load i32, i32* @__llvm_prof_foo_csid, align 4
  %0 = mul i32 %load_csid, 1
  %1 = add i32 %0, 0
  %2 = getelementptr inbounds [5 x i64], [5 x i64]* @__profc_foo, i32 0, i32 %1
  %pgocount = load i64, i64* %2, align 8
  %3 = add i64 %pgocount, 1
  store i64 %3, i64* %2, align 8
  ret void
}
```
Implementation of callsite-aware PGO

- Example (IR Dump After Frontend instrumentation-based coverage lowering)

```assembly
define dso_local void @foo() #0 {
  entry:
  %load_csid = load i32, i32* @_llvm_prof_foo_csid, align 4
  %0 = mul i32 %load_csid, 1
  %1 = add i32 %0, 0
  %2 = getelementptr inbounds [5 x i64], [5 x i64]* @_profc_foo, i32 0, i32 %1
  %pgocount = load i64, i64* %2, align 8
  %3 = add i64 %pgocount, 1
  store i64 %3, i64* %2, align 8
  ret void
}
```
\textbf{llvm-profdatal}

- Overview of this tool
- What possibilities does it have
  - show
  - merge
  - overlap
- How to add an extension to it
  - \texttt{llvm/tools/llvm-profdatal/llvm-profdatal.cpp}
  - \texttt{lib/ProfileData/InstrProfReader.cpp}
  - \texttt{lib/ProfileData/InstrProfWriter.cpp}
Clang and profdata

• How to load profdata from file
  • -fprofile-use=/path/to/merged/profdata
  • lib/ProfileData/InstrProf.cpp
    • readNextRecord
    • readHeader
    • readRawCounts
    • ReadData
    • createSymtab
Clang and profdata

- Where to store it
- lib/Transforms/Instrumentation/PGOInstrumentation.cpp
  - FuncPGOInstrumentation
  - MST
  - Assign branch-probabilities to BBs
- How this data can be used by optimizations?
  - different inline strategy for different callsites
Summary

- PGO is an important optimization approach
- LLVM PGO can be easily extended
- Still it has a lot of work to do
- Now you are able to enhance PGO in LLVM
  - Add something new or
  - Fix existing features. There is a pair (of dozens) of TODOs and FIXMEs in PGO-related files (e.g. need to add support for instrument select instructions, which uses condition with vector type)
Thank you!
Q&A
Links

Patch:

• https://github.com/kpdev/llvm-project/tree/llvm-dev-mtg/callsite

Docs:

• PGO Docs:  https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

Presentations:

• MSVC team talk: https://channel9.msdn.com/Shows/C9-GoingNative/C9GoingNative-12-C-at-BUILD-2012-Inside-Profile-Guided-Optimization