Introducing Function Specialization, and can we enable it by default?

Sjoerd Meijer

Sjoerd.meijer@arm.com

US LLVM Developer conference 2021

Introducing Function Specialization

- Inter-procedural optimization (IPO),
  - New LLVM IR transformation pass (off by default).
- Improve runtime performance, at the expense of:
  - Compile-time,
  - Code-size.
- It improves:
  - MCF in the SPEC benchmark, but also
  - Is general so that it triggers e.g. in the LLVM test-suite, stage2 builds, etc.
- GCC has this enabled by default at -O3, so we're missing out...
- It lives in: *llvm/lib/Transforms/IPO/FunctionSpecialization.cpp*
  - First commit reviewed in D93838,
  - Based on previous work in D36432 by Matthew Simpson.

Motivating Example

```
int foo(int x, int flag) {
  if (flag)
    return compute(x, plus);
  return compute(x, minus);
}
static int compute(int x, int (*binop)(int)) {
  return binop(x);
}
static int plus(int x) {
  return x + 1;
}
static int minus(int x) {
  return x - 1;
}
```

- **Problem**: a lot of indirect calls.
  - Can we optimise this?
  - Can we promote indirect calls to direct?

- **Solution**:
  - Look at functions and its arguments.
  - Propagate constant args down to its func body
  - Constant args = constant globals, functions.

Motivating Example, cont'd

Input

```
int foo(int x, int flag) {
  if (flag)
    return compute(x, plus);
  return compute(x, minus);
}
static int compute(int x, int (*binop)(int)) {
  return binop(x);
}
static int plus(int x) {
  return x + 1;
}
static int minus(int x) {
  return x - 1;
}
```

Specialize *compute()*

on constant arg *binop.*

Output

```
int foo(int x, int flag) {
  if (flag)
    return compute.1(x);
  return compute.2(x);
}
static int compute.1(int x) {
  return plus(x);
}
static int compute.2(int x) {
  return minus(x);
}
static int plus(int x) {  return x + 1; }
static int minus(int x) {  return x - 1; }
```

Motivating Example, cont'd

- Then, the direct call(s) get inlined further:

```
int foo(int x, int flag) {
  if (flag)
    return x + 1;
  return x - 1;
}
```

- **Observation**: isn't this a roundabout way of doing inlining?
- Maybe, but by design:
  - FuncSpec is run before the inliner in the optimisation pipeline.
  - Otherwise, we would only benefit from constant passing (TODO).

Inlining vs. Function Specialisation

- **Inlining**:
  - Natural place if inlining is the goal?
- Cons:
  - Inlining heuristics are difficult already.
  - Specialising would require a whole new infrastructure on top of that.

- **FuncSpec**:
  - Relatively straightforward pass (to implement).
  - GCC has function specialization enabled at O3 ("if GCC can do it").
  - Supports different use cases: i) inlining functions, ii) propagating integer constant (ranges).
- Cons:
  - Increases compile-times and code-size more?

Cost-model

- Goal-oriented heuristic: estimate if replacing an argument with a particular constant value would result in optimization opportunities

- if SpecializationBonus(Arg) > SpecializationCost(F), then Profitable!

- SpecializationCost(F) = F.NumInst * InstrCost * NbFuncSpec

- SpecializationBonus(Arg) =
  - For all uses of Arg: add the instruction cost, scaled by the loopnest depth.
  - For all call-sites: get the inline cost, add this to the instruction cost

Compile-time Results CTMark

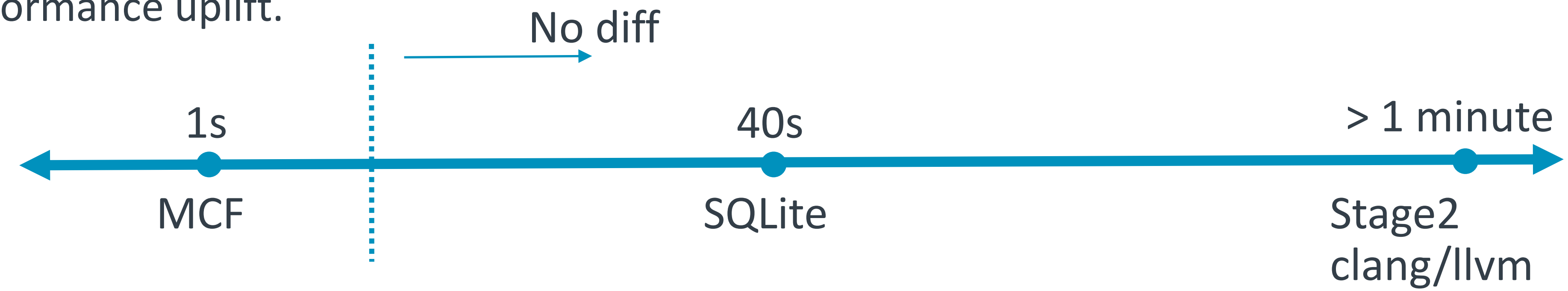
Program	% Increase	# FS	Forced
kimwitu++	+0.12	0	0
sqlite3	+0.32	0	111
consumer-typeset	-0.07	0	1
Bullet	+0.29	0	1
tramp3d-v4	+0.28	0	0
mafft	+0.49	0	0
ClamAV	**+0.39**	**2**	**24**
lencod	+0.45	0	0
SPASS	+0.36	0	55
7zip	+0.12	0	4
Geomean	**+0.28**		

- LLVM compile-time-tracker
  - Wall clock time can be noisy,
  - Retired # instruction proxy for compile-times
  - O3, ReleaseThinLTO, ReleaseLTO-g and O0-g

- -O3 and -flto: triggers 2x in ClamAV

Compile-times, cont'd

- Wall clock times can be stable.
- Clang/LLVM Stage2 build & SQLite:
  - 3 functions specialised,
  - No difference in compile-times.
- MCF (SPEC2017):
  - 2 functions specialised,
  - 20% compile-time increase (LTO link-step),
  - 10% performance uplift.



- Little time spent in pass FuncSpec
- Backend processes more functions/instructions
- Bigger impact on smaller compile jobs, less on bigger.

Future Work

- Can we enable FuncSpec by default?
- Add ThinLTO support.
- Cost-model:
  - Constant integers are support, but not enabled.
  - To avoid too many specialisations, only 1 argument per function is specialised.
  - Comp-times are not suggesting this, but analysis results are not cached.
- Introduce an attribute/pragma to explicitly request specialisation.

Feedback welcome!

- LLVM dev mailing list
- Phabricator
- Direct email

Click to add text

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

شكرًا

תודה