# Shrink-wrapping for separate components

Segher Boessenkool
segher@kernel.crashing.org

GNU Tools Cauldron, Prague, September 2017

## Shrink-wrapping, in general

The prologue and epilogue handle many things that need to be done before (or after) a function executes.

But some of those things are not needed by all of the function, only by some parts of it.

Shrink-wrapping moves (pieces of) the prologue and epilogue to places where they are executed less often, making sure they are still around every piece of code that needs it.

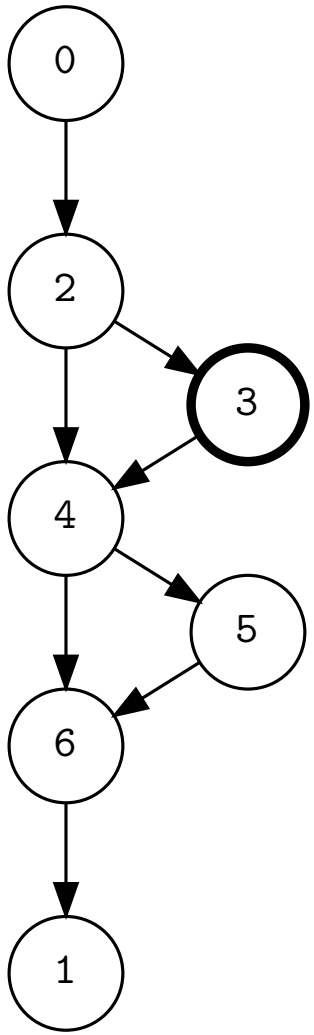
# 3 Shrink-wrapping, in GCC

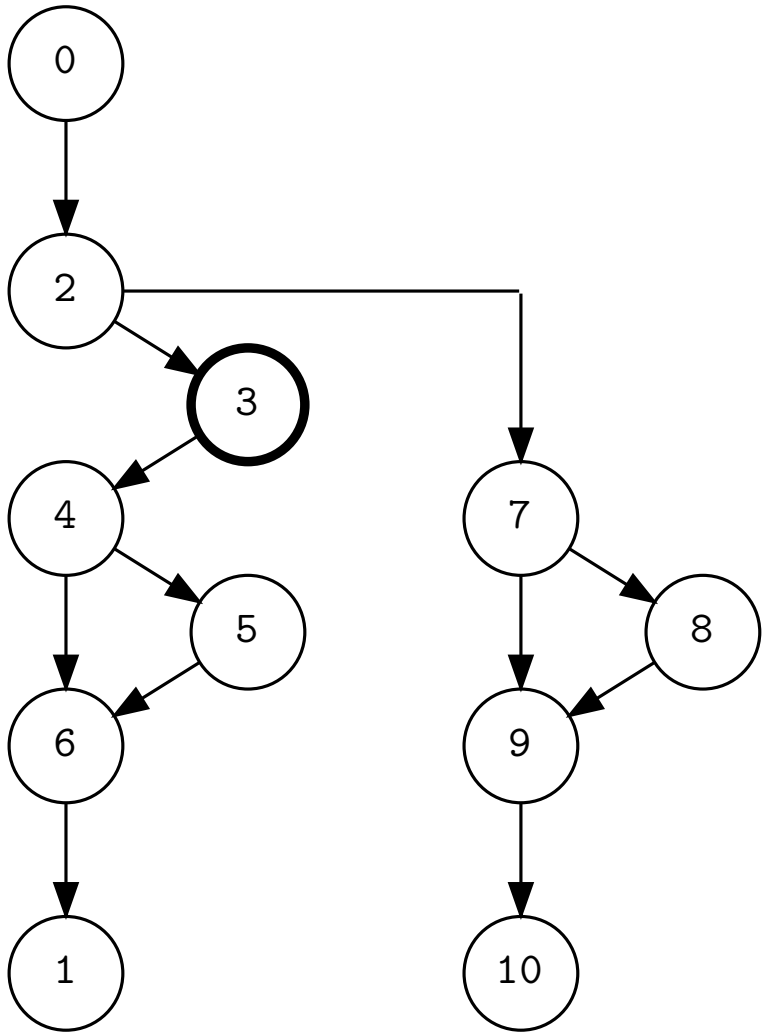
In GCC shrink-wrapping is done in `shrink-wrap.c` (via `function.c`).

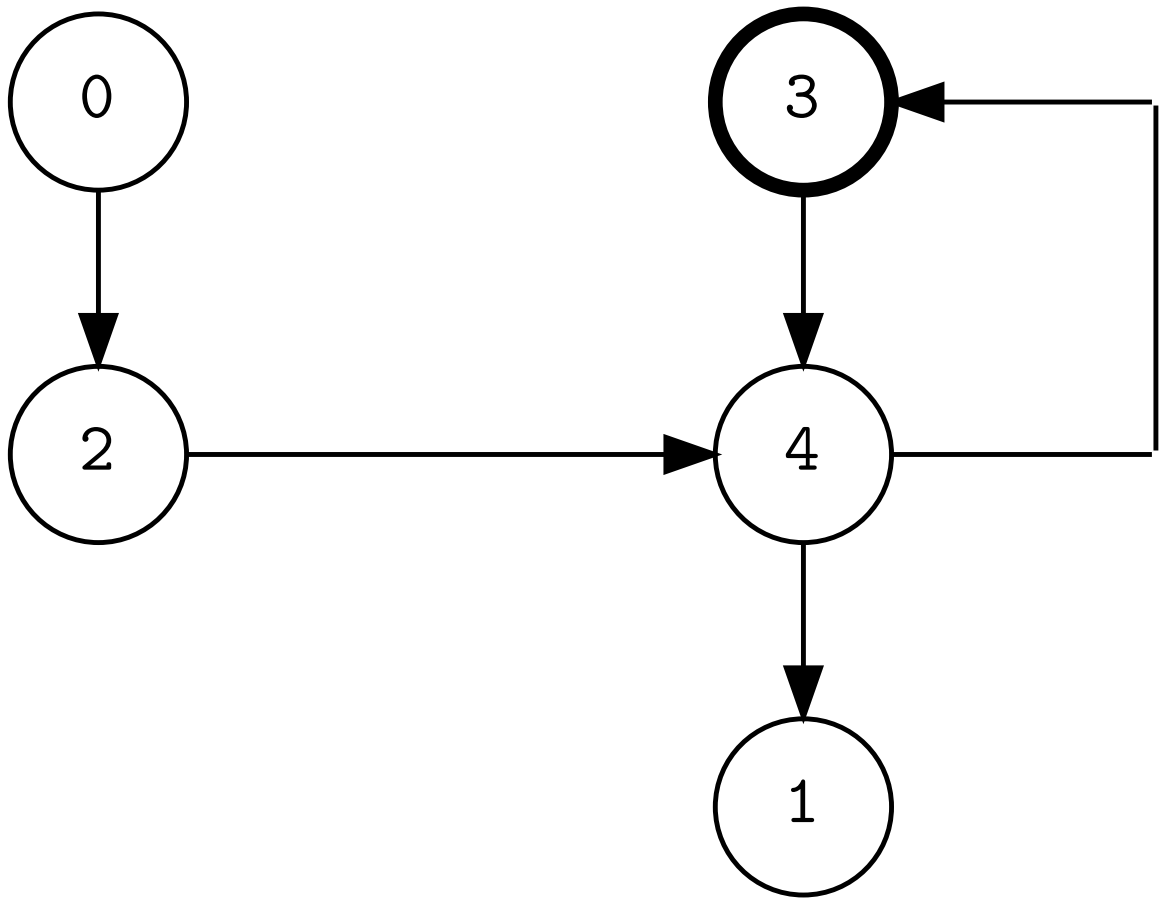
A restriction in GCC is that the prologue and epilogue code can be emitted only once per function.

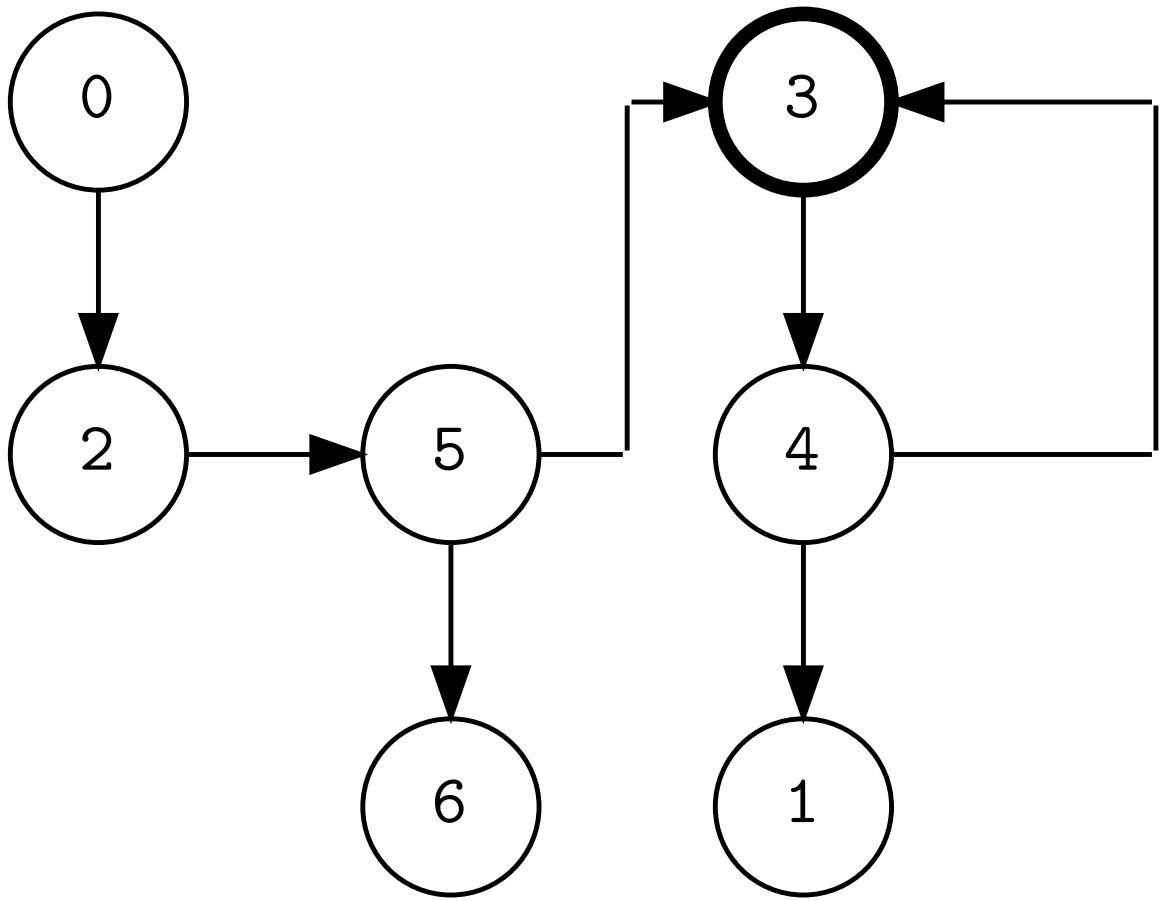
Some basic blocks do not need anything the prologue sets up.

If some paths through the function do not need the prologue at all, the shrink-wrapping code makes that happen, duplicating blocks where it needs to.


0
2
3
4
5
6
1


0
2
3
4
5
6
1
7
8
9
10


0
3
2
4
1


0
3
2
5
4
6
1

## Shrink-wrapping for separate components

The prologue code can be big, do many things, for example on machines that have many (callee-saved) registers.

Most of those things are independent of each other. Most of those things would work fine if the prologue/epilogue pair was executed more than once. Most of those things aren't needed everywhere.

The separate shrink-wrapping code handles those things (which it calls "components") separate from the rest of the prologue/epilogue.

## 9 Hooks

The generic separate shrink-wrapping code knows nothing about what the components really represent; that is all up to the target backend code.

Each GCC target that wants to use separate shrink-wrapping needs to implement some hooks in the `shrink_wrap` group.

`get_separate_components`, to tell what components exist.

`components_for_bb`, to say what components are needed by each basic block.

## More hooks

`emit_prologue_components` and `emit_epilogue_components`, to actually emit code for those components.

`disqualify_components` is a way for the backend to tell the generic code that it should not emit some prologue (or epilogue) component on a given edge.

`set_handled_components` tells the backend what components are actually handled by the separate shrink-wrapping code, so that they can be left out of the normal prologue/epilogue.

# 11 How to place things

The goal of separate shrink-wrapping is to make every component executed as infrequently as possible.

We know (or have guessed) the `bb->frequency` for each basic block, so we use that info.

We can place (the prologues/epilogues for) every component separately.

However, trying all possible placements is computationally infeasible.

And copying blocks can lead to exponential code growth.

## How to place things (2)

One option is to place prologue/epilogue components only on the entry and exit of (maximal) single-entry single-exit regions.

Such regions are always nested, never overlapping, making it easy to find the optimal placement of the components.

This is quite simple and elegant, except that finding SESE regions is not.

At this stage in the GCC pass pipeline we also do not generally have as many SESE regions as you would hope.

## How to place things (3)

Unless a function does not return, the epilogue components will be executed exactly as often as the prologue components. So, for computing the optimal placement we can ignore the epilogue components.

Call a component "active" on some basic block if that block will be executed with the prologue for that component having been executed more often than the epilogue. We first determine which components are active, for every block, and then later place prologues and epilogues to make that true.

Initialise this to what each block strictly needs.

# How to place things (4)

If a component is active in some block, we can make it active in all blocks dominated by that first block, for no cost.

Now set an `own_cost` for each block, as its execution frequency minus the frequency of the backedges to that block.

Do a depth-first search on the dominator tree.

If a block's `own_cost` is less than the sum of the costs of its children, it should be made active itself, because that is cheaper to execute.

## How to place things (5)

This procedure overestimates the cost when jumped to from a non-dominating block.

This does not seem to matter in practice; it results in good placement for everything I have seen.

For SESE regions it gives exactly the same results as the algorithm a few slides back.

```
void g(void);

void f(int x)
{
  register int r20 asm("20") = x;
  asm("" : : "r"(r20));
  if (x)
    g();
  asm("");
}
```

```
        mflr 0              # save LR
        std 20,-96(1)       # save r20
        mr 20,3
        std 0,16(1)         # save LR
        stdu 1,-208(1)      # set up stack frame
        cmpdi 7,3,0
        beq 7,L2
        bl g
        nop
L2:
        addi 1,1,208        # tear down stack frame
        ld 0,16(1)          # restore LR
        ld 20,-96(1)        # restore r20
        mtlr 0              # restore LR
        blr
```

```
void g(void);

void f(int x)
{
  // main prologue is inserted here
  if (x) {
    // save LR here
    g();
    // restore LR here
  }
}
```

```
        std 20,-96(1)     # save r20
        stdu 1,-208(1)    # set up stack frame
        mr 20,3
        cmpdi 7,3,0
        beq 7,L2
        mflr 0            # save LR
        std 0,224(1)      # save LR
        bl g
        nop
        ld 0,224(1)       # restore LR
        mtlr 0            # restore LR
L2:
        addi 1,1,208      # tear down stack frame
        ld 20,-96(1)      # restore r20
        blr
```

```
void g(void) __attribute__((noreturn));
void h(void) __attribute__((noreturn));

void f(int x)
{
  if (x == 42)
    g();
  if (x == 31)
    h();
}
```

```
void f(int x)
{
  if (x == 42) {
    // save LR here
    g();
    // unreachable here, g does not return
  }
  if (x == 31) {
    // save LR here
    h();
    // unreachable here, h does not return
  }
}
```

```
int *a;
void g(void);

void f(void)
{
  int j;
  for (j = 0; j < 4; j++) {
    if (__builtin_expect(a[j], 0))
      g();
    asm("#" : : : "memory");
    if (__builtin_expect(a[j], 0))
      g();
    a[j]++;
  }
}
```

```
void f(void)
{
  int j;
  // inserting LR save here makes it cost 1
  for (j = 0; j < 4; j++) {
    if ( ... )
      g(); // 10% of the time
    if ( ... )
      g(); // 10% of the time
  }
  // inserting LR save/restore around the
  // calls to g costs only 0.8
}
```

## Benchmark results

SPECint 2006 improves by ∼2%

- perlbench ∼2%
- gcc ∼2%
- hmmer ∼8%
- astar ∼6%
- xalancbmk ∼2%

SPECfp 2006 improves by ∼1.5%

- povray ∼15%

## Benchmark results (2)

SPECint 2017 improves by ∼0.2%

- perlbench ∼2%
- gcc ∼2%
- mcf ∼1%
- omnetpp ∼2%
- xalancbmk degrades
- xz degrades

## Benchmark results (3)

Also building glibc with `-fshrink-wrap-separate`:

SPECint 2017 improves by ∼0.6%

- perlbench ∼3%
- gcc ∼2%
- mcf ∼1%
- omnetpp ∼3%
- xalancbmk still degrades (but less)
- xz improves

# Benchmark results (4)

SPECfp 2017 improves by ∼2%

- povray ∼13%
- blender ∼2%
- imagick ∼13%

PHP micro-benchmarks improve by up to 35%

## Future work

For rs6000, we should have more components: for the VRs, the CR fields, the TOC pointer, ...

We should be able to express dependencies between components, so that for example we can have a component for the frame setup, or stack alignment.

## Future work (2)

We need some way for shrink-wrapping to do live range splitting or similar. This will increase the opportunities for separate shrink-wrapping about threefold.

Shrink-wrapping currently tries to do this a little bit, moving register copies from the very first block to later.

Also the RA tries, see `split_live_ranges_for_shrink_wrap`, but it isn't currently very effective (it only deals with argument registers copied into pseudos, for example).

# Backup

## Some details

After determining which blocks we definitely do want to have some component active, we make it active on as many blocks as we can without making it active on any block on any path from entry to exit where it wasn't already active.

This gives more natural placement, and saves some code space.

We don't put separate prologues there where the "main" prologue will go: the target code often knows how to make the main prologue code more optimal.

## More details

If we need to place some prologue or epilogue on an edge that cannot be split, we have a problem. In that case, completely give up separately wrapping the components in question.

Similarly, if the backend says it does not want to handle it (for example, if it wants to use some hard register as temporary, but that register is already live on that edge).

## Even more details

If we need a prologue on all incoming edges to a block, place the prologue directly in the block instead of on the edges. Similarly for outgoing edges.

The cross-jumping pass would try to do this, but it often fails to do it.

This saves some code space.