Function Multi Versioning for AArch64: Compiler aided function specialization with runtime dispatch

Alexandros Lamprineas

Intro

The problem

- Software is deployed on various devices (we may compile for a baseline and reuse the binaries).
- Most CPUs implement optional instructions which may not be present on the target of compilation.
- For example, on Arm, dotproduct instructions may not be available.
- To make use of such instructions, a run-time check is needed.

The solution

Function Multi Versioning (FMV)
https://learn.arm.com/learning-paths/cross-platform/function-multiversioning

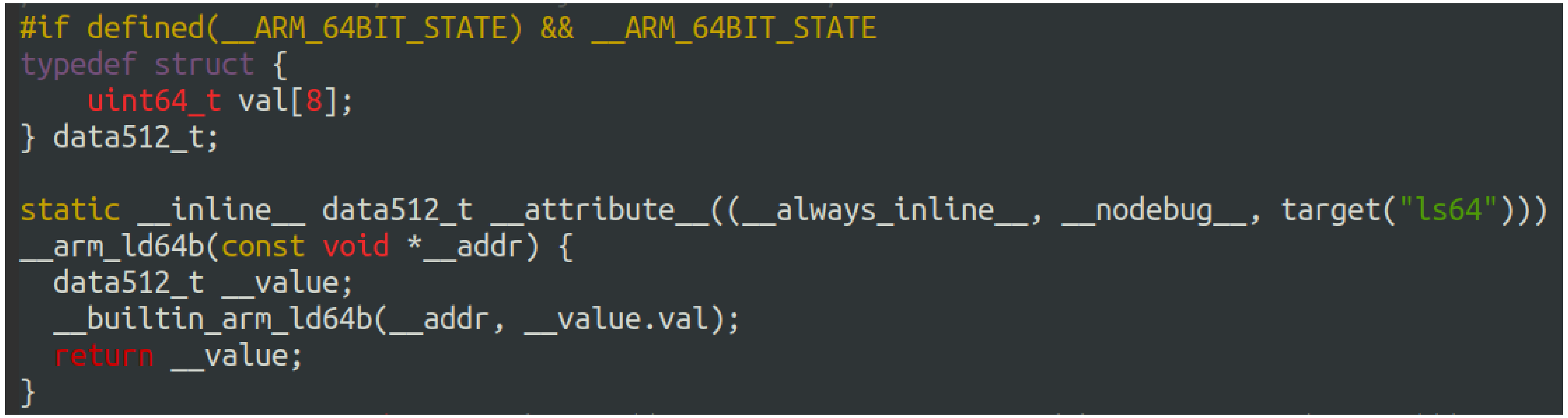
- lets the compiler generate multiple function versions, and
- auto-dispatch between them

What’s new?

- FMV was originally developed for x86 using the function attributes ***target*** and ***target_clones***
  - https://llvm.org/devmtg/2014-10/#talk22 `__attribute__((target("sse4.2")))`
  - https://reviews.llvm.org/D40819

- On AArch64 the ***target*** attribute is broadly used as an optimization hint,
  - for example in header files of Arm C Language Extensions:

lib/clang/21/include/arm_acle.h



```
#if defined(__ARM_64BIT_STATE) && __ARM_64BIT_STATE
typedef struct {
    uint64_t val[8];
} data512_t;

static __inline__ data512_t __attribute__((__always_inline__, __nodebug__, target("ls64")))
__arm_ld64b(const void *__addr) {
  data512_t __value;
  __builtin_arm_ld64b(__addr, __value.val);
  return __value;
}
```

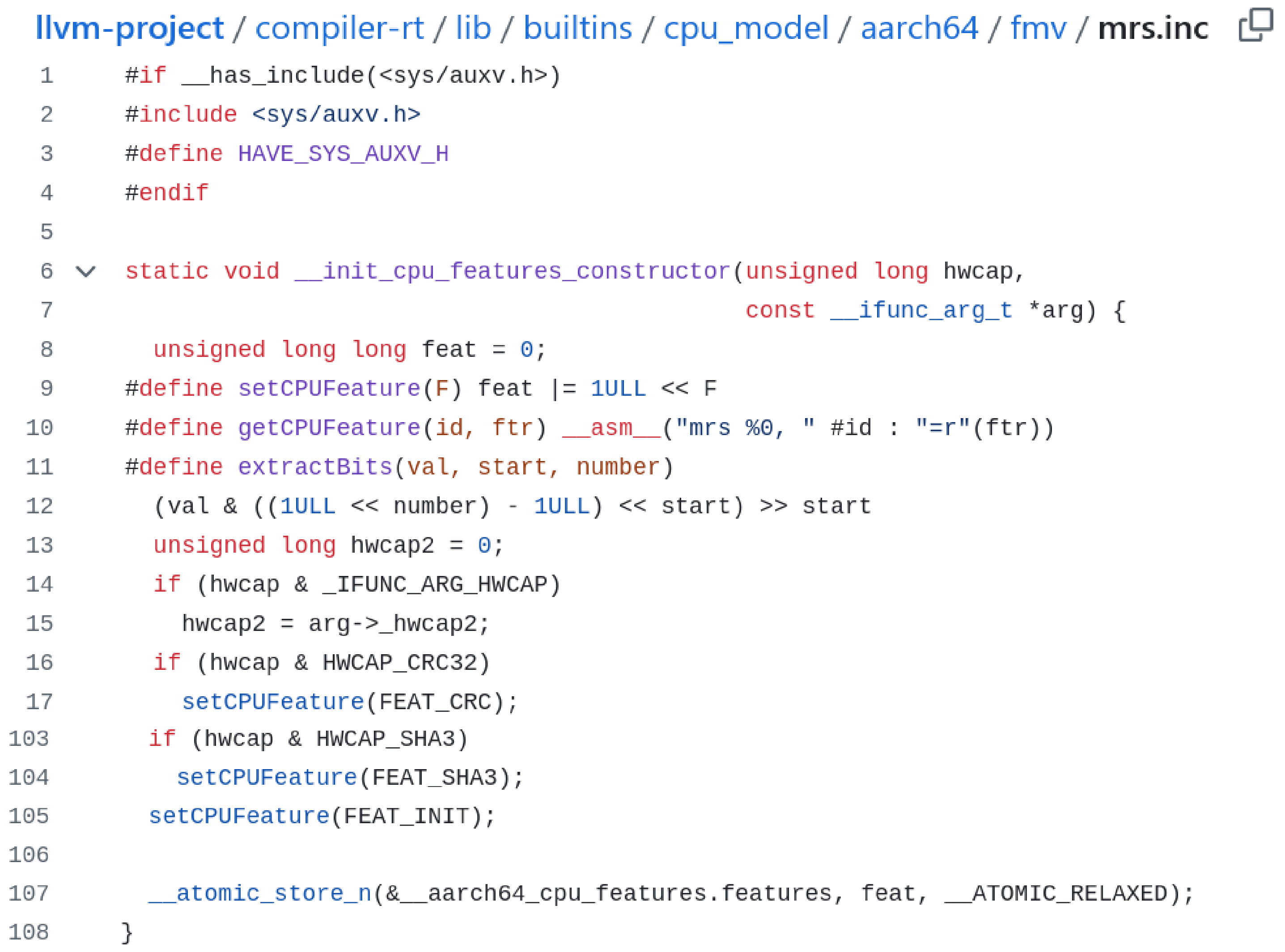
- Therefore, we introduced a new attribute ***target_version*** (also adopted by RISC-V) and wrote a specification for FMV in ACLE (currently beta).

Clang CodeGen

- Clang generates multiple function versions with ***mangled names*** as per https://arm-software.github.io/acle/main/acle.html#name-mangling
  - each version may use several features, their names are encoded in the mangled name
  - `__attribute((target_version("crc+bti+aes+bf16"))) int fmv(void);` → `@fmv._MaesMbf16MbtiMcrc()`
- Each version is associated with ***metadata*** which propagate information from C/C++ source to LLVM.
- Clang generates a ***resolver function*** that determines which version to run when the function is called.
  - The resolution is dynamic (it is performed at load time) and permanent for the lifetime of the process.
  - The resolver: (1) initializes the runtime, (2) detects features, and (3) selects the available version of highest priority as indicated by https://arm-software.github.io/acle/main/acle.html#mapping
- Clang generates a global ***__aarch64_cpu_features*** which contains the feature bits for runtime detection.

Compiler-rt

The runtime asks the kernel which features are available on host and initializes *__aarch64_cpu_features*



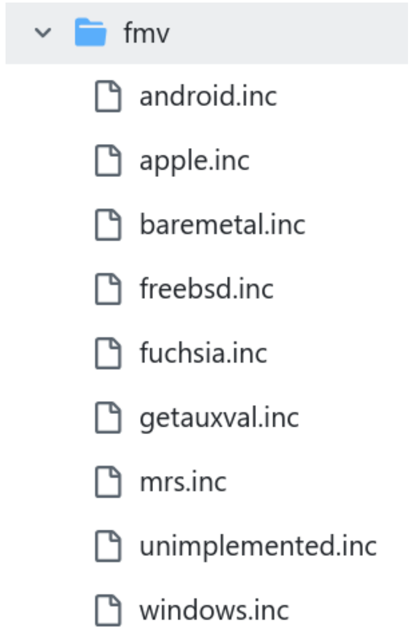
llvm-project / compiler-rt / lib / builtins / cpu_model / aarch64 / fmv / **mrs.inc**

```
#if __has_include(<sys/auxv.h>)
#include <sys/auxv.h>
#define HAVE_SYS_AUXV_H
#endif

static void __init_cpu_features_constructor(unsigned long hwcap,
                                            const __ifunc_arg_t *arg) {
  unsigned long long feat = 0;
#define setCPUFeature(F) feat |= 1ULL << F
#define getCPUFeature(id, ftr) __asm__("mrs %0, " #id : "=r"(ftr))
#define extractBits(val, start, number)
  (val & ((1ULL << number) - 1ULL) << start) >> start
  unsigned long hwcap2 = 0;
  if (hwcap & _IFUNC_ARG_HWCAP)
    hwcap2 = arg->_hwcap2;
  if (hwcap & HWCAP_CRC32)
    setCPUFeature(FEAT_CRC);
  if (hwcap & HWCAP_SHA3)
    setCPUFeature(FEAT_SHA3);
  setCPUFeature(FEAT_INIT);

  __atomic_store_n(&__aarch64_cpu_features.features, feat, __ATOMIC_RELAXED);
}
```

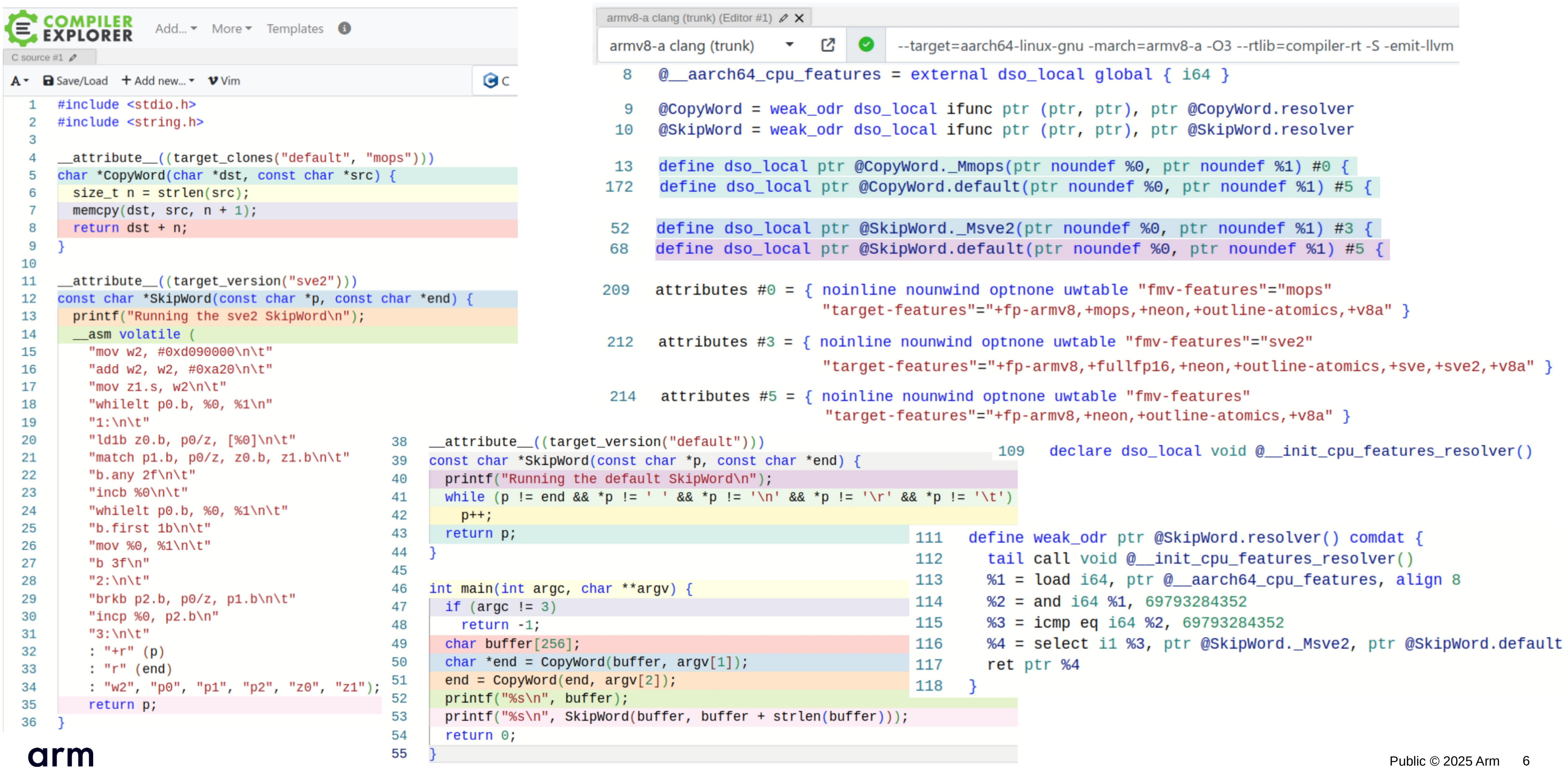
Various platforms are supported



```
fmv
  android.inc
  apple.inc
  baremetal.inc
  freebsd.inc
  fuchsia.inc
  getauxval.inc
  mrs.inc
  unimplemented.inc
  windows.inc
```

Example

https://godbolt.org/z/qsaTohfM7



```c
#include <stdio.h>
#include <string.h>

__attribute__((target_clones("default", "mops")))
char *CopyWord(char *dst, const char *src) {
  size_t n = strlen(src);
  memcpy(dst, src, n + 1);
  return dst + n;
}

__attribute__((target_version("sve2")))
const char *SkipWord(const char *p, const char *end) {
  printf("Running the sve2 SkipWord\n");
  __asm volatile (
    "mov w2, #0xd090000\n\t"
    "add w2, w2, #0xa20\n\t"
    "mov z1.s, w2\n\t"
    "whilelt p0.b, %0, %1\n"
    "1:\n\t"
    "ld1b z0.b, p0/z, [%0]\n\t"
    "match p1.b, p0/z, z0.b, z1.b\n\t"
    "b.any 2f\n\t"
    "incb %0\n\t"
    "whilelt p0.b, %0, %1\n\t"
    "b.first 1b\n\t"
    "mov %0, %1\n\t"
    "b 3f\n"
    "2:\n\t"
    "brkb p2.b, p0/z, p1.b\n\t"
    "incp %0, p2.b\n"
    "3:\n\t"
    : "+r" (p)
    : "r" (end)
    : "w2", "p0", "p1", "p2", "z0", "z1");
    return p;
}
```

```c
__attribute__((target_version("default")))
const char *SkipWord(const char *p, const char *end) {
  printf("Running the default SkipWord\n");
  while (p != end && *p != ' ' && *p != '\n' && *p != '\r' && *p != '\t')
    p++;
  return p;
}

int main(int argc, char **argv) {
  if (argc != 3)
    return -1;
  char buffer[256];
  char *end = CopyWord(buffer, argv[1]);
  end = CopyWord(end, argv[2]);
  printf("%s\n", buffer);
  printf("%s\n", SkipWord(buffer, buffer + strlen(buffer)));
  return 0;
}
```

```
--target=aarch64-linux-gnu -march=armv8-a -O3 --rtlib=compiler-rt -S -emit-llvm
```

```llvm
@__aarch64_cpu_features = external dso_local global { i64 }

@CopyWord = weak_odr dso_local ifunc ptr (ptr, ptr), ptr @CopyWord.resolver
@SkipWord = weak_odr dso_local ifunc ptr (ptr, ptr), ptr @SkipWord.resolver

define dso_local ptr @CopyWord._Mmops(ptr noundef %0, ptr noundef %1) #0 {
define dso_local ptr @CopyWord.default(ptr noundef %0, ptr noundef %1) #5 {

define dso_local ptr @SkipWord._Msve2(ptr noundef %0, ptr noundef %1) #3 {
define dso_local ptr @SkipWord.default(ptr noundef %0, ptr noundef %1) #5 {

attributes #0 = { noinline nounwind optnone uwtable "fmv-features"="mops"
                  "target-features"="+fp-armv8,+mops,+neon,+outline-atomics,+v8a" }

attributes #3 = { noinline nounwind optnone uwtable "fmv-features"="sve2"
                  "target-features"="+fp-armv8,+fullfp16,+neon,+outline-atomics,+sve,+sve2,+v8a" }

attributes #5 = { noinline nounwind optnone uwtable "fmv-features"
                  "target-features"="+fp-armv8,+neon,+outline-atomics,+v8a" }
```

```llvm
declare dso_local void @__init_cpu_features_resolver()
```

```llvm
define weak_odr ptr @SkipWord.resolver() comdat {
  tail call void @__init_cpu_features_resolver()
  %1 = load i64, ptr @__aarch64_cpu_features, align 8
  %2 = and i64 %1, 69793284352
  %3 = icmp eq i64 %2, 69793284352
  %4 = select i1 %3, ptr @SkipWord._Msve2, ptr @SkipWord.default
  ret ptr %4
}
```

Design choices

Resolver emission

- FMV is supported across multiple translation units.
- The resolver cannot “see” versions beyond the current translation unit.
- Emission options:
  - On use (when the function is called)? → may generate multiple (potentially different) resolvers
    → non deterministic version selection depending on linking order ✖
  - Unique resolver in the TU of the default version
    → deterministic version selection regardless of linking order ✔

Feature detection

- Dependent-on features get detected transitively as indicated by https://arm-software.github.io/acle/main/acle.html#dependencies (for example **sve2**→ **sve** → **fp16** → **fp**)
- Features implied by the command line are not exempt from runtime detection (**simd** → **fp**)
  - For example -march=armv8 implies **simd**

FMV info representation

FMV info is autogenerated using
llvm/utils/TableGen/ArmTargetDefEmitter.cpp
https://github.com/llvm/llvm-project/pull/113281

llvm-project / llvm / include / llvm / TargetParser / **AArch64CPUFeatures.inc**

```
// Function Multi Versioning CPU features.
enum CPUFeatures {
  FEAT_RNG,
  FEAT_FLAGM,
  FEAT_FLAGM2,
  FEAT_FP16FML,
  FEAT_DOTPROD,
  FEAT_SM4,
  FEAT_RDM,
  FEAT_LSE,
  FEAT_FP,
  FEAT_SIMD,
  FEAT_CRC,
  RESERVED_FEAT_SHA1, // previously used and now ABI legacy
  FEAT_SHA2,
```

- ***detection ≠ priority*** because the detection bit is part of the ABI; if a feature is added/removed whose priority falls between existing ones…

llvm-project / llvm / include / llvm / TargetParser / **AArch64TargetParser.h**

```
struct FMVInfo {
  StringRef Name;                // The target_version/target_clones spelling.
  CPUFeatures FeatureBit;        // Index of the bit in the FMV feature bitset.
  FeatPriorities PriorityBit;    // Index of the bit in the FMV priority bitset.
  std::optional<ArchExtKind> ID; // The architecture extension to enable.
```

llvm-project / llvm / include / llvm / TargetParser / **AArch64FeatPriorities.inc**

```
// Function Multi Versioning feature priorities.
enum FeatPriorities {
  PRIOR_RNG,
  PRIOR_FLAGM,
  PRIOR_FLAGM2,
  PRIOR_LSE,
  PRIOR_FP,
```

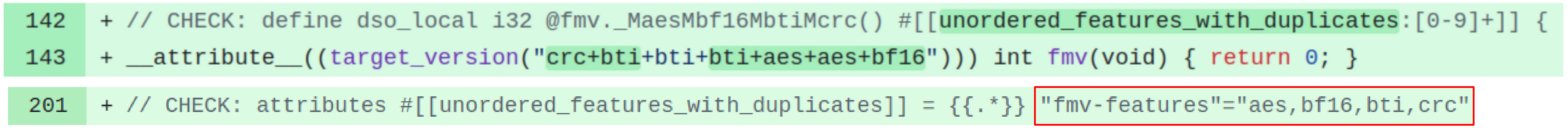
```
// Represents a dependency between two architecture extensions. Later is the
// feature which was added to the architecture after Earlier, and expands the
// functionality provided by it. If Later is enabled, then Earlier will also be
// enabled. If Earlier is disabled, then Later will also be disabled.
struct ExtensionDependency {
  ArchExtKind Earlier;
  ArchExtKind Later;
};
```

- ***dependencies*** are used both for (1) runtime detection, and (2) to enable all the necessary subtarget features for code generation

Metadata in LLVM IR

- https://github.com/llvm/llvm-project/pull/118544
  Similar to ***target-features***.

clang/test/CodeGen/AArch64/fmv-features.c



```
+ // CHECK: define dso_local i32 @fmv._MaesMbf16MbtiMcrc() #[[unordered_features_with_duplicates:[0-9]+]] {
+ __attribute__((target_version("crc+bti+bti+bti+aes+aes+bf16"))) int fmv(void) { return 0; }
```

```
+ // CHECK: attributes #[[unordered_features_with_duplicates]] = {{.*}} "fmv-features"="aes,bf16,bti,crc"
```

Why we need them?

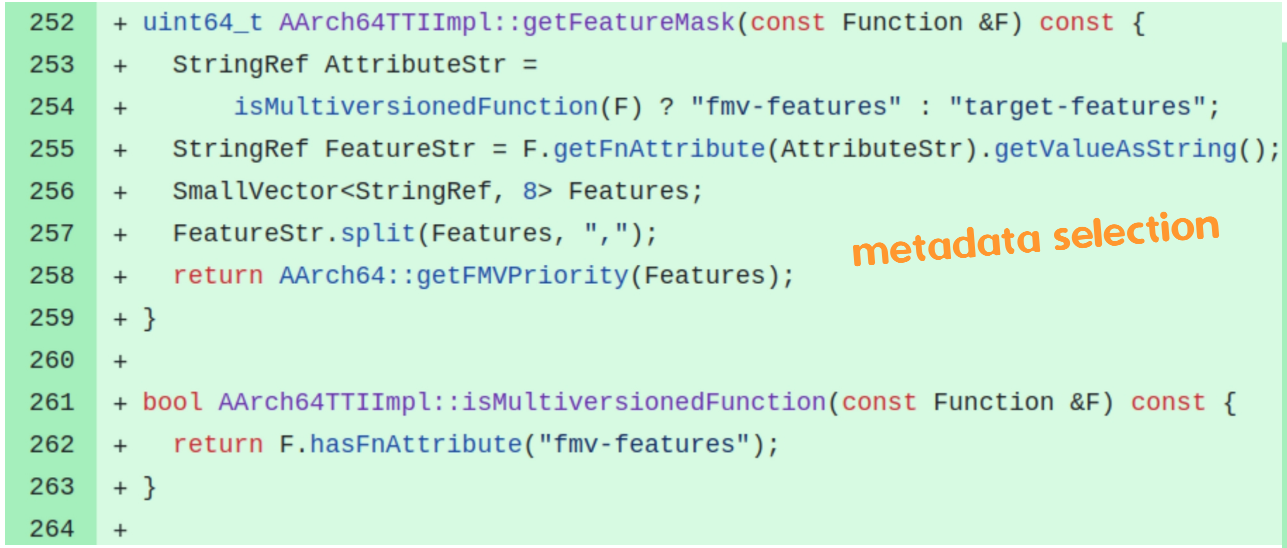
- Suppose you have **target_version("i8mm+dotprod")** and **target_version("fcma").**
- The first version has higher priority because Priority(i8mm) > Priority(fcma) > Priority(dotprod).
- Now suppose you specify **-march=armv8-a+i8mm** on the command line.
- Then the versions would have **target-features** "+dotprod,+i8mm" and "+fcma,+i8mm" respectively.
- If you are using these metadata to deduce version priority, then you would incorrectly deduce that the second version was higher priority than the first!

GlobalOpt

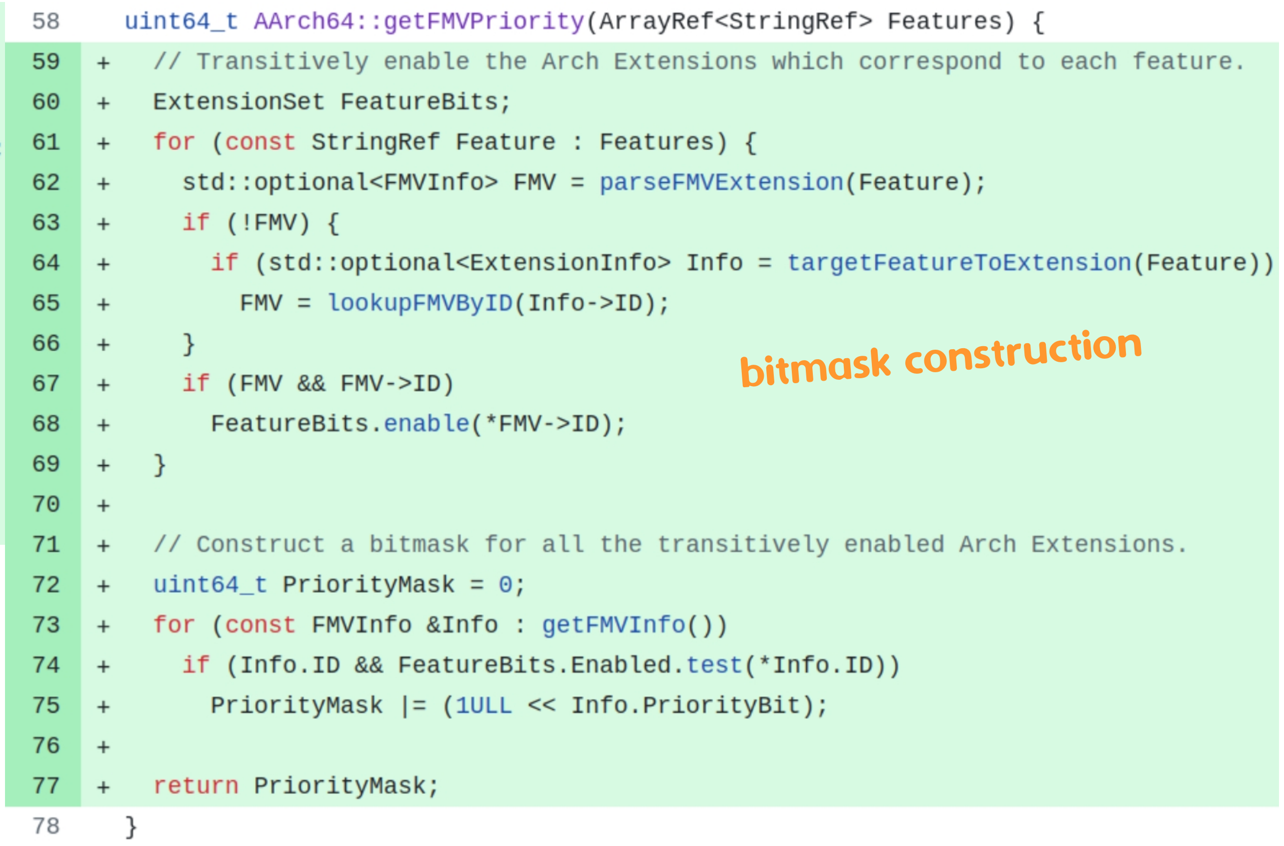
- May statically (at compile time) resolve calls to versioned functions
  https://github.com/llvm/llvm-project/pull/87939

  by comparing LLVM IR metadata between caller and callee.

benefit? → inlining



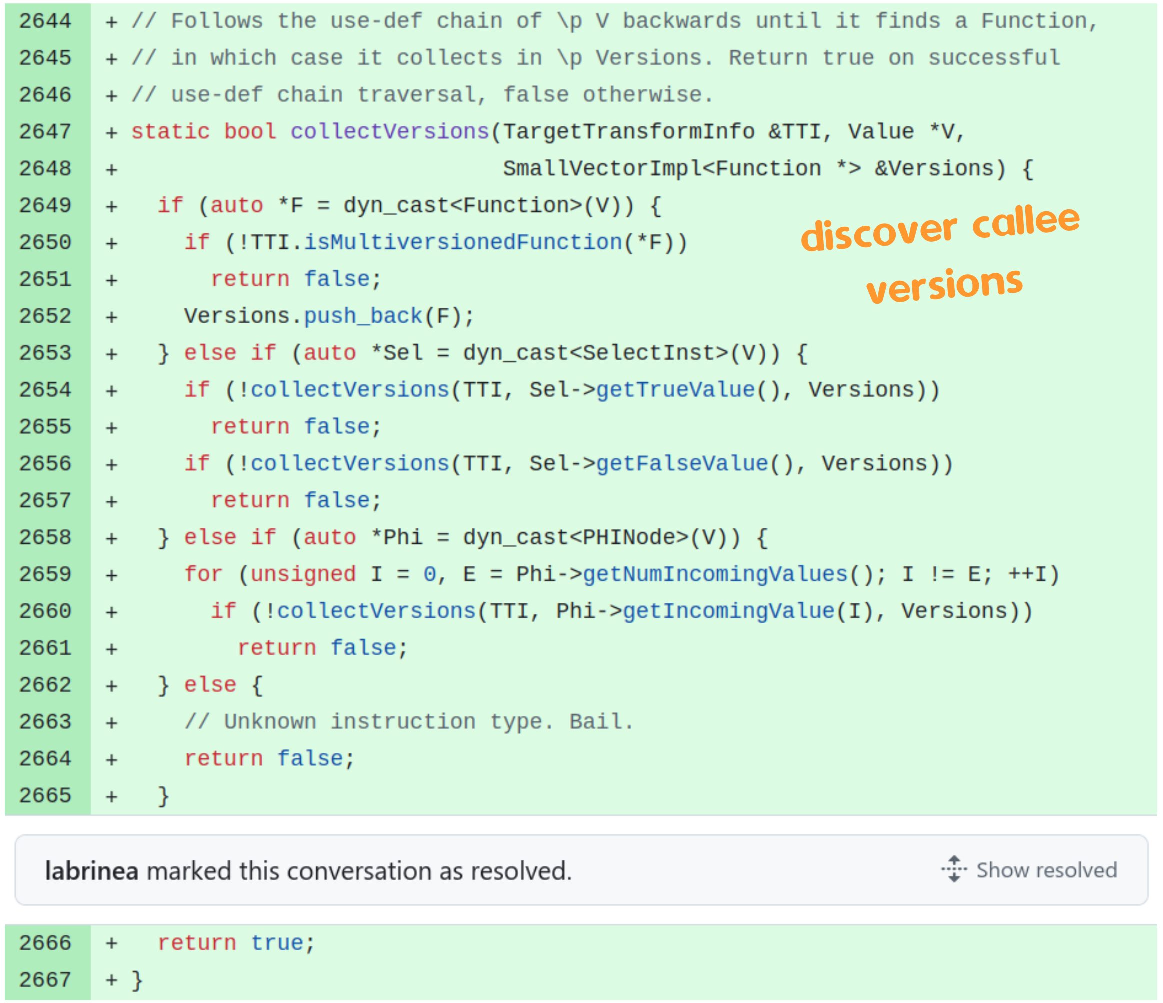
```
+ uint64_t AArch64TTIImpl::getFeatureMask(const Function &F) const {
+   StringRef AttributeStr =
+       isMultiversionedFunction(F) ? "fmv-features" : "target-features";
+   StringRef FeatureStr = F.getFnAttribute(AttributeStr).getValueAsString();
+   SmallVector<StringRef, 8> Features;
+   FeatureStr.split(Features, ",");
+   return AArch64::getFMVPriority(Features);
+ }
+
+ bool AArch64TTIImpl::isMultiversionedFunction(const Function &F) const {
+   return F.hasFnAttribute("fmv-features");
+ }
+
```



```
  uint64_t AArch64::getFMVPriority(ArrayRef<StringRef> Features) {
+   // Transitively enable the Arch Extensions which correspond to each feature.
+   ExtensionSet FeatureBits;
+   for (const StringRef Feature : Features) {
+     std::optional<FMVInfo> FMV = parseFMVExtension(Feature);
+     if (!FMV) {
+       if (std::optional<ExtensionInfo> Info = targetFeatureToExtension(Feature))
+         FMV = lookupFMVByID(Info->ID);
+     }
+     if (FMV && FMV->ID)
+       FeatureBits.enable(*FMV->ID);
+   }
+
+   // Construct a bitmask for all the transitively enabled Arch Extensions.
+   uint64_t PriorityMask = 0;
+   for (const FMVInfo &Info : getFMVInfo())
+     if (Info.ID && FeatureBits.Enabled.test(*Info.ID))
+       PriorityMask |= (1ULL << Info.PriorityBit);
+
+   return PriorityMask;
  }
```

- if FMV caller → FMV callee,
  then compare **fmv-features**
- else if non-FMV caller → FMV callee,
  then compare **target-features** with **fmv-features**

Static resolution algorithm

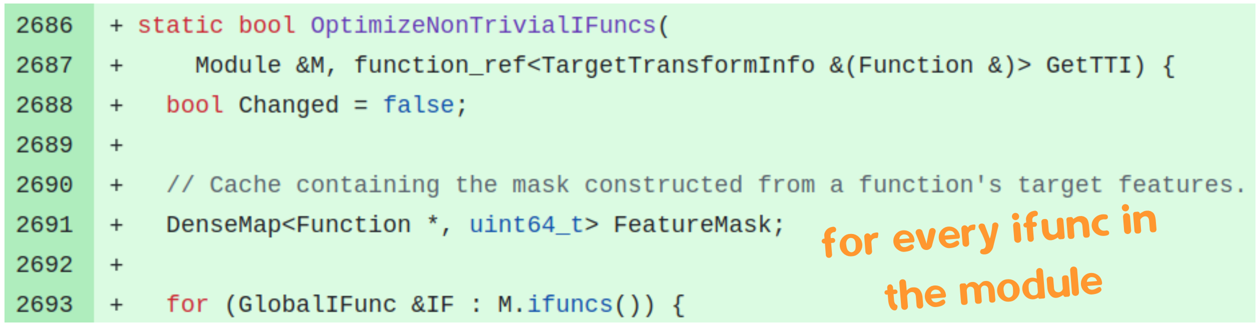


```
+ // Follows the use-def chain of \p V backwards until it finds a Function,
+ // in which case it collects in \p Versions. Return true on successful
+ // use-def chain traversal, false otherwise.
+ static bool collectVersions(TargetTransformInfo &TTI, Value *V,
+                             SmallVectorImpl<Function *> &Versions) {
+   if (auto *F = dyn_cast<Function>(V)) {
+     if (!TTI.isMultiversionedFunction(*F))
+       return false;
+     Versions.push_back(F);
+   } else if (auto *Sel = dyn_cast<SelectInst>(V)) {
+     if (!collectVersions(TTI, Sel->getTrueValue(), Versions))
+       return false;
+     if (!collectVersions(TTI, Sel->getFalseValue(), Versions))
+       return false;
+   } else if (auto *Phi = dyn_cast<PHINode>(V)) {
+     for (unsigned I = 0, E = Phi->getNumIncomingValues(); I != E; ++I)
+       if (!collectVersions(TTI, Phi->getIncomingValue(I), Versions))
+         return false;
+   } else {
+     // Unknown instruction type. Bail.
+     return false;
+   }
```

discover callee versions

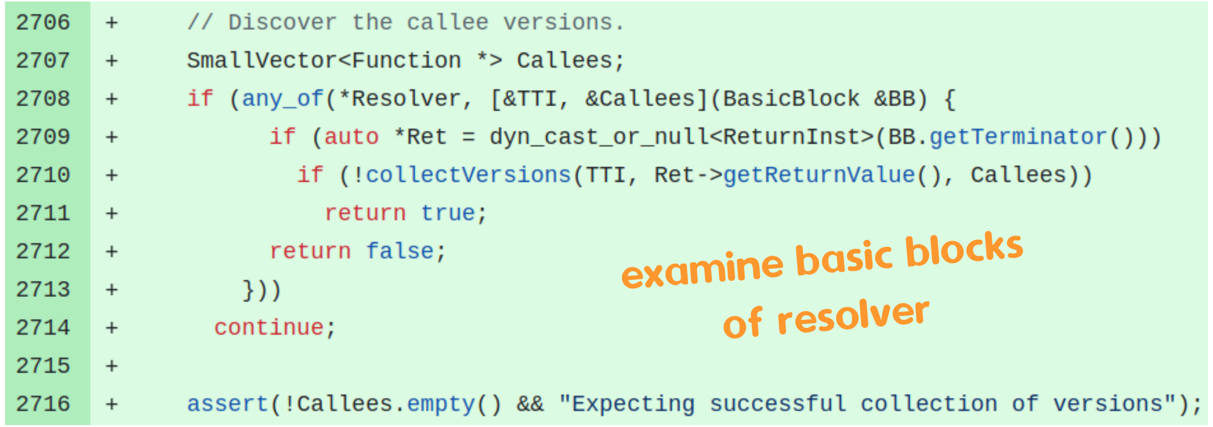
labrinea marked this conversation as resolved. Show resolved

```
+   return true;
+ }
```



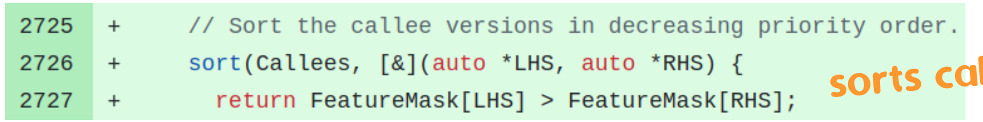
```
+ static bool OptimizeNonTrivialIFuncs(
+     Module &M, function_ref<TargetTransformInfo &(Function &)> GetTTI) {
+   bool Changed = false;
+
+   // Cache containing the mask constructed from a function's target features.
+   DenseMap<Function *, uint64_t> FeatureMask;
+
+   for (GlobalIFunc &IF : M.ifuncs()) {
```

for every ifunc in the module



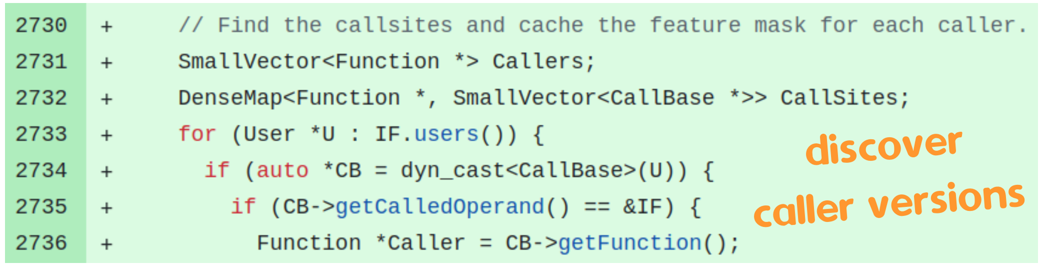
```
+     // Discover the callee versions.
+     SmallVector<Function *> Callees;
+     if (any_of(*Resolver, [&TTI, &Callees](BasicBlock &BB) {
+           if (auto *Ret = dyn_cast_or_null<ReturnInst>(BB.getTerminator()))
+             if (!collectVersions(TTI, Ret->getReturnValue(), Callees))
+               return true;
+           return false;
+         }))
+       continue;
+
+     assert(!Callees.empty() && "Expecting successful collection of versions");
```

examine basic blocks of resolver



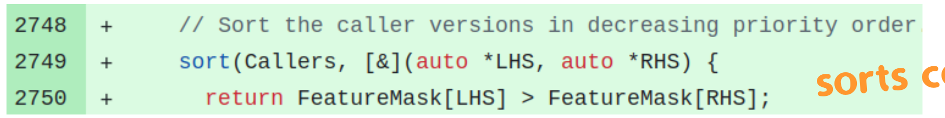
```
+     // Sort the callee versions in decreasing priority order.
+     sort(Callees, [&](auto *LHS, auto *RHS) {
+       return FeatureMask[LHS] > FeatureMask[RHS];
```

sorts callees



```
+     // Find the callsites and cache the feature mask for each caller.
+     SmallVector<Function *> Callers;
+     DenseMap<Function *, SmallVector<CallBase *>> CallSites;
+     for (User *U : IF.users()) {
+       if (auto *CB = dyn_cast<CallBase>(U)) {
+         if (CB->getCalledOperand() == &IF) {
+           Function *Caller = CB->getFunction();
```

discover caller versions

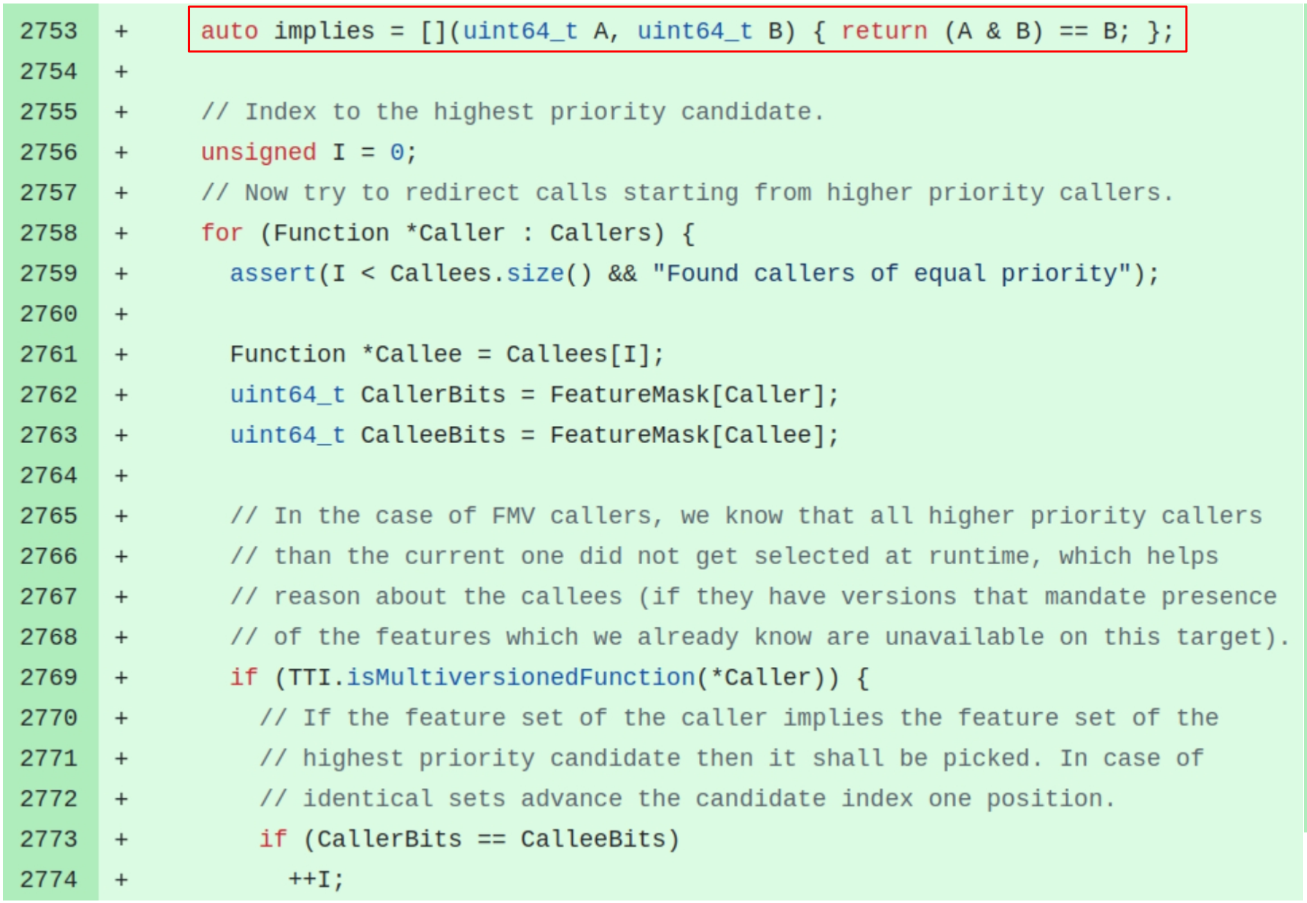


```
+     // Sort the caller versions in decreasing priority order.
+     sort(Callers, [&](auto *LHS, auto *RHS) {
+       return FeatureMask[LHS] > FeatureMask[RHS];
```

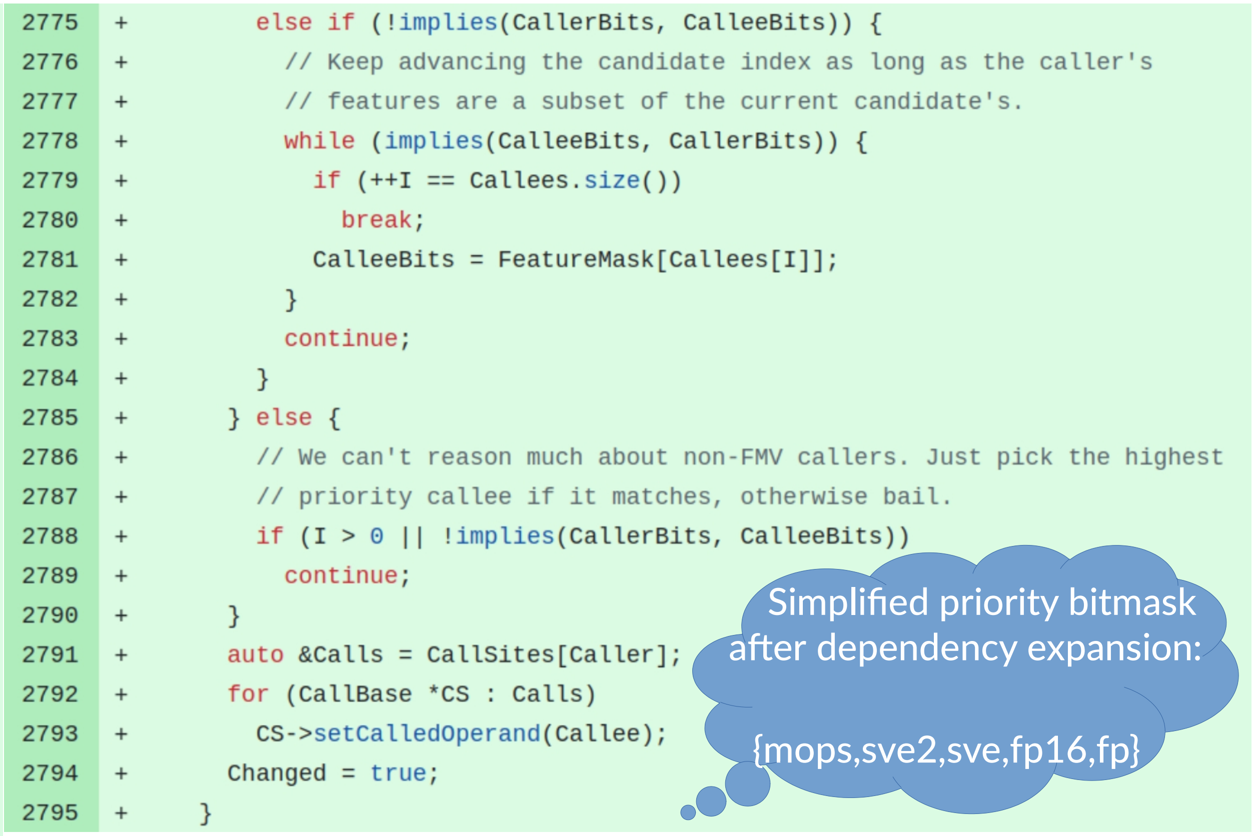
sorts callers

Static resolution algorithm (continued)

Example https://godbolt.org/z/PcGnGbWd3
from llvm/test/Transforms/GlobalOpt/resolve-fmv-ifunc.ll



```
+    auto implies = [](uint64_t A, uint64_t B) { return (A & B) == B; };
+
+    // Index to the highest priority candidate.
+    unsigned I = 0;
+    // Now try to redirect calls starting from higher priority callers.
+    for (Function *Caller : Callers) {
+      assert(I < Callees.size() && "Found callers of equal priority");
+
+      Function *Callee = Callees[I];
+      uint64_t CallerBits = FeatureMask[Caller];
+      uint64_t CalleeBits = FeatureMask[Callee];
+
+      // In the case of FMV callers, we know that all higher priority callers
+      // than the current one did not get selected at runtime, which helps
+      // reason about the callees (if they have versions that mandate presence
+      // of the features which we already know are unavailable on this target).
+      if (TTI.isMultiversionedFunction(*Caller)) {
+        // If the feature set of the caller implies the feature set of the
+        // highest priority candidate then it shall be picked. In case of
+        // identical sets advance the candidate index one position.
+        if (CallerBits == CalleeBits)
+          ++I;
```



```
+        else if (!implies(CallerBits, CalleeBits)) {
+          // Keep advancing the candidate index as long as the caller's
+          // features are a subset of the current candidate's.
+          while (implies(CalleeBits, CallerBits)) {
+            if (++I == Callees.size())
+              break;
+            CalleeBits = FeatureMask[Callees[I]];
+          }
+          continue;
+        }
+      } else {
+        // We can't reason much about non-FMV callers. Just pick the highest
+        // priority callee if it matches, otherwise bail.
+        if (I > 0 || !implies(CallerBits, CalleeBits))
+          continue;
+      }
+      auto &Calls = CallSites[Caller];
+      for (CallBase *CS : Calls)
+        CS->setCalledOperand(Callee);
+      Changed = true;
+    }
```

Simplified priority bitmask after dependency expansion:

{mops,sve2,sve,fp16,fp}

sorted caller versions | **sorted callee versions**

caller._MmopsMsve2 = {1,1,1,1,1} → callee._Mmops = {1,0,0,0,0}
caller._Mmops = {1,0,0,0,0} → callee._Mmops = {1,0,0,0,0}
caller._Msve = {0,0,1,1,1} | callee._Msve2 = {0,1,1,1,1}
caller.default = {0,0,0,0,0} → callee.default = {0,0,0,0,0}
callee._Msve = {0,0,1,1,1}

- mops+sve2 implies mops → we can statically resolve
- mops implies mops → we can statically resolve
  bitmask equality → advance callee iterator
- at this point we know the host does not have mops
  sve does not imply sve2 → we can't statically resolve
  however sve2 implies sve → advance callee iterator
- keep skipping over callee candidates
  sve implies sve → advance callee iterator
- no feature is available → we can statically resolve

arm

Future work

- User can control feature priorities: https://github.com/ARM-software/acle/pull/371
- User can refer to a specific function version: https://github.com/llvm/llvm-project/issues/84094
- Pointer authentication works with IFUNC resolver: https://github.com/llvm/llvm-project/pull/84704
- Request to support more features (like CSSC): https://github.com/llvm/llvm-project/issues/131218
- ? (feedback welcome)

Acknowledgements

Special thanks to all the folks who helped with ACLE/code reviews, code refactoring, technical discussions, brainstorming, presentation, etc.

Jon Roelofs, Tomas Matheson, Andrew Carlotti, Daniel Kiss, Wilco Dijkstra, Victor Campos, Sander De Smalen, Maciej Gabka, Andre Vieira, Richard Sandiford, Pavel Iliin, Kristof Beyls, Alfie Richards and others.

arm

Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank You
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm

arm