High-performance Load-time Implementation Selection

JOE BIALEK, PRANAV KANT

Cppcon
The C++ Conference
2022
September 12th-16th

Welcome to CppCon 2022!

Join #visual_studio channel on CppCon Discord
https://aka.ms/cppcon/discord

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
https://aka.ms/cppcon

Problem Space

A function such as “memcpy” needs to have a unique implementation per-CPU.

Or some CPU’s get their own implementation and others get a more generic implementation based on general features supported by the CPU such as AVX, AVX2, etc.

Memcpy runs VERY fast – need zero-overhead mechanism to switch between implementations.

How is this Handled Today?

All techniques evaluated in the context of "how would this look on Windows", not necessarily "how is it implemented on other OS's".

Developer Initialized and Checked Program State

Developers write some logic that happens when binary is initializing after load. Checks CPU features, etc. Sets up global variables.

Within hot functions like memcpy, developer uses this state to choose the most optimal code sequence.

Examples:

- Indirect call a function pointer that was initialized
- Test-and-branch pattern to select best function

Performance Issues

1. Indirect calls are EXPENSIVE in kernel-mode due to speculative execution mitigations (clearing branch predictor state).

2. Indirect calls are EXPENSIVE because of CFI (Control-Flow-Integrity) checks (i.e. Control Flow Guard, Clang-CFI).

3. Test-and-branch’s cost scales with the number of checks (can easily end up costing more than the function being optimized).

4. Making global state writeable, then read-only, is expensive (TLB shootdowns). Affects binary load time.

Usability Issues

Developer needs to manually initialize and check state.

Hard to initialize state generically (i.e. what if this is a static CRT that gets linked in to code in user-mode, kernel-mode, boot loader, etc.)?

Some environments (like kernel-mode) don't call CRT initializers. Need to do even more custom stuff.

GCC IFunc Approach

Solves a lot of usability issues. For a function like “memcpy” being optimized, a “selector” function is defined.

When code tries to call the function for the first time, the selector function gets called instead. Selector function runs developer-defined code and returns a pointer to the most optimal implementation to use (i.e. a “memcpy” specific to your CPU model).

Issues - GCC IFunc Approach

Involves making an indirect call (performance).

If CFI checks are omitted for perf, creates a security issue.

Single “selector” function makes it difficult for multiple libraries to contribute specializations for the same base function (like memcpy).

Note: If IFunc is used for a DLL export, there is no perf penalty for callers of the DLL Export. This function call is already done indirectly and IFunc simply “updates” the exported functions address.

Load Time
Function Selection

Requirements

Friendly, flexible developer model.

Excellent performance – Retain direct calls and jumps, no indirect calls or cascading branches.

Maintains One Definition Rule.

Secure.

Extensible without requiring compiler updates.

Building Blocks

1. OS defined “capabilities”.
2. Binary metadata format to map “set of capabilities present” and “function to use”.
3. Compiler syntax to create metadata.
4. OS support to parse metadata and use it to modify binary code pages based on result.

1. OS Capabilities

OS Capabilities (overridecapabilities.h in SDK)

```
enum {
  OVRDCAP_AMD64_ERMSB                  = 0x00000000
  OVRDCAP_AMD64_FAST_SHORT_REPMOV      = 0x00000001
  OVRDCAP_AMD64_FAST_ZERO_LEN_REPMOV   = 0x00000002
  …
  OVRDCAP_AMD64_V1_CAPSET              = 0x0000013B
  …
};
```

Currently these are exposed as macros but they will be an enum.

Versioning: V1_CAPSET indicates OS knows about all previous capabilities.

Capabilities allow querying:

- Specific CPU Features or CPU Model
- Operating System Features
- Anything else people need ☺

OS can publish new capabilities without updating the compiler.

2. Binary Metadata

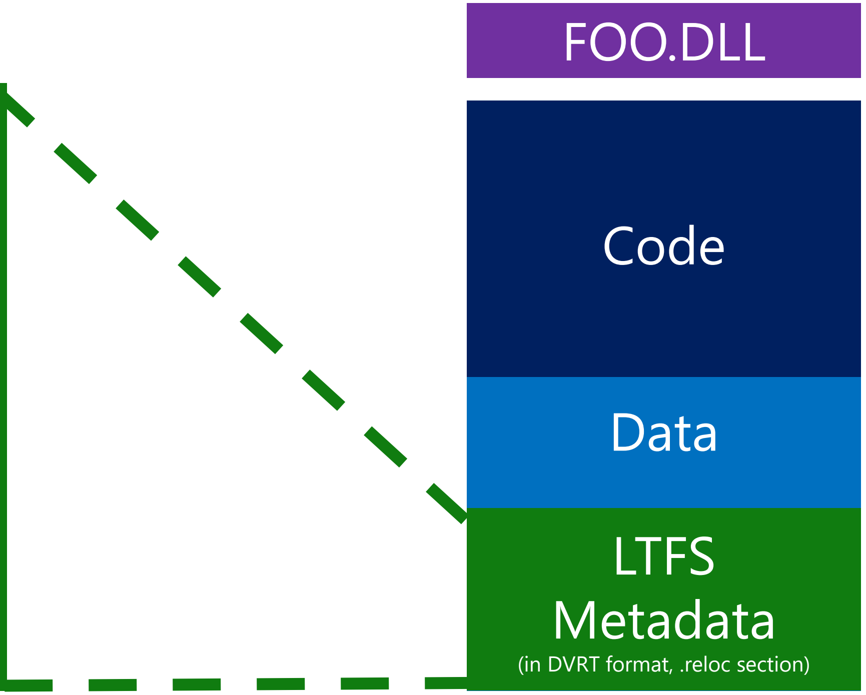
Load Time Function Selection (LTFS) Metadata

For each function receiving Load Time Function Selection:

BDD (Binary Decision Diagram): Represents evaluation criteria.

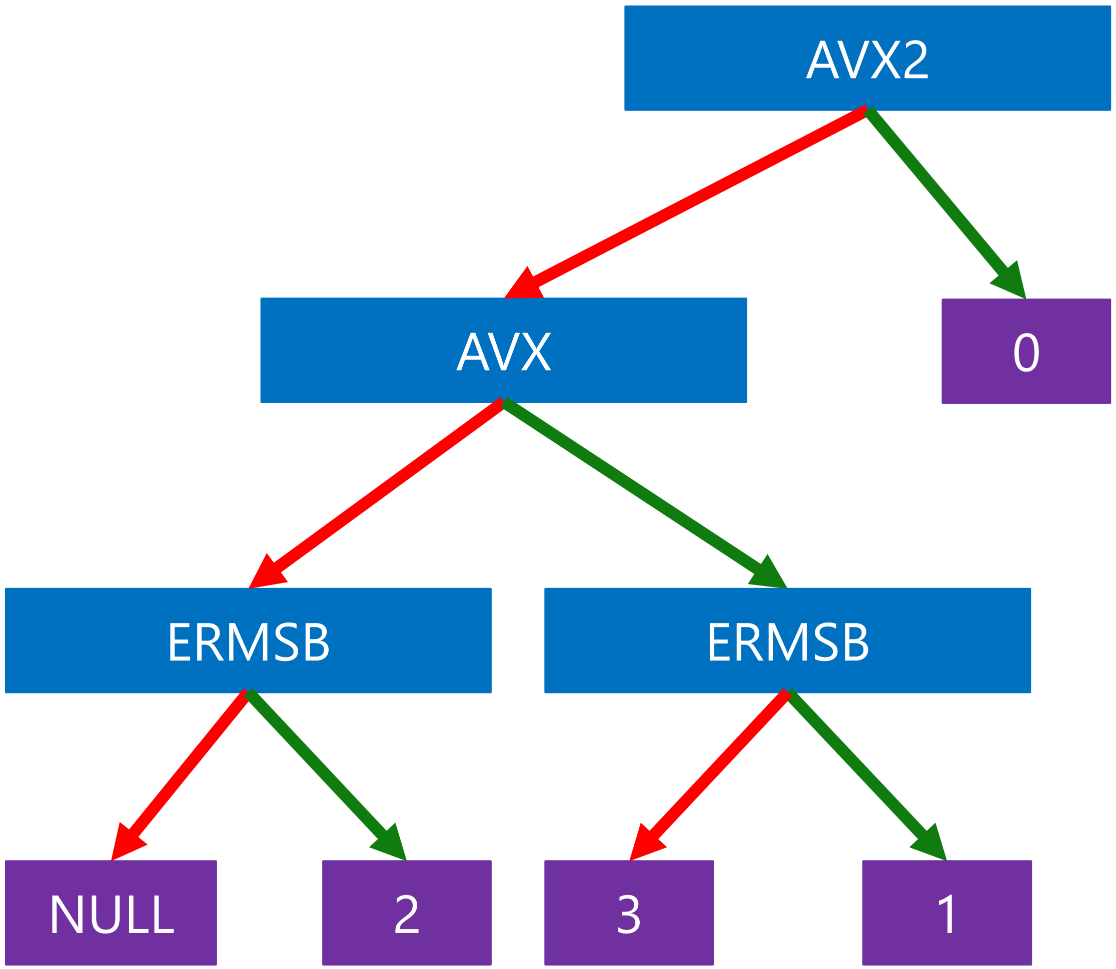
RVAs: Array containing RVA of each candidate function.

Fixup RVAs: Array containing RVA for every location in the binary that direct calls/jumps into the target function.



RVA (Relative Virtual Address): Offset from the start of the binary

Binary Decision Diagram (BDD)



Each node in the BDD either a decision node or a leaf node.

Decision nodes contain a capability to check, and the next node to evaluate if this capability is true/false.

Leaf nodes contain either:
1) The index into the table that contains the RVA of the selected function.
2) NULL node that indicates no specialized function found.

RVAs array:

Index	Function RVA
0	0xB000
1	0xB150
2	0xA090
3	0x5000

3. Compiler Syntax

Open Questions Remain!

Scenarios to Support

Specify a set of ordered evaluation criteria

“if (FOO and BAR) use FooBar”, “if (FOO) use Foo”, “if (BAR) use Bar”

Function naming flexibility.

If candidate function implemented in C/ASM, developer can manually specify its name.

If function is implemented in C++, function can be named automatically using name mangling.

Merge evaluation criteria from multiple TU’s in a sensible way.

Concepts

Qualifier

- Get automatically sorted relative to one another by the linker.

Ordered Map

- Gets put into binary in the precise order specified by the developer.

Function MUST be dispatch attributed in all TU's to use this feature.

Qualifier and Ordered Map can be used together or independently.

Use the same set of OS Capabilities.

Qualifier Rules

LTFS criteria can use any number of capabilities in their qualifier (including none).

The order in which qualifiers are seen by the linker doesn’t matter. The order in which capabilities are listed in a qualifier doesn’t matter.

The capability with the smallest numerical constant value gets evaluated first.

“Most specific” qualifier goes first. If one qualifier is “ERMSB” and the other is “AVX and ERMSB”, the latter is the most specific.

Qualifier Examples

TU 1:

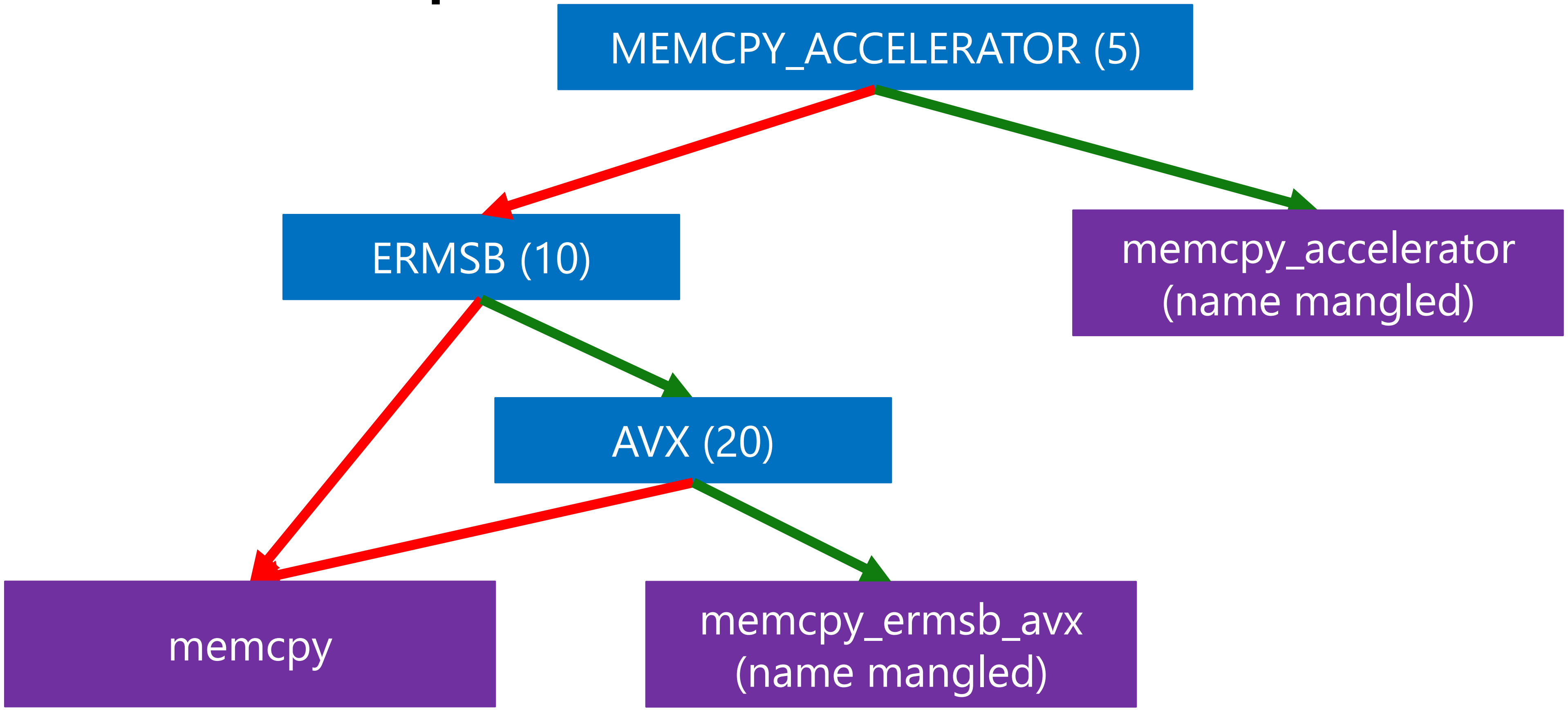
```
[[msvc::dispatch(qualifier: [all(AVX, ERMSB)])]]
void* memcpy(void*, void*, size_t);
```

TU 2:

```
[[msvc::dispatch(qualifier: [all(MEMCPY_ACCELERATOR)])]]
void* memcpy(void*, void*, size_t);
```

Assume numerical value as follows: AVX (20) > ERMSB (10) > MEMCPY_ACCELERATOR (5)

Qualifier Examples



Order in which capabilities are evaluated based on their numerical value, sorted by linker.

Function names are mangled by compiler (but friendly names displayed here for clarity).

Qualifier Examples

TU 1:

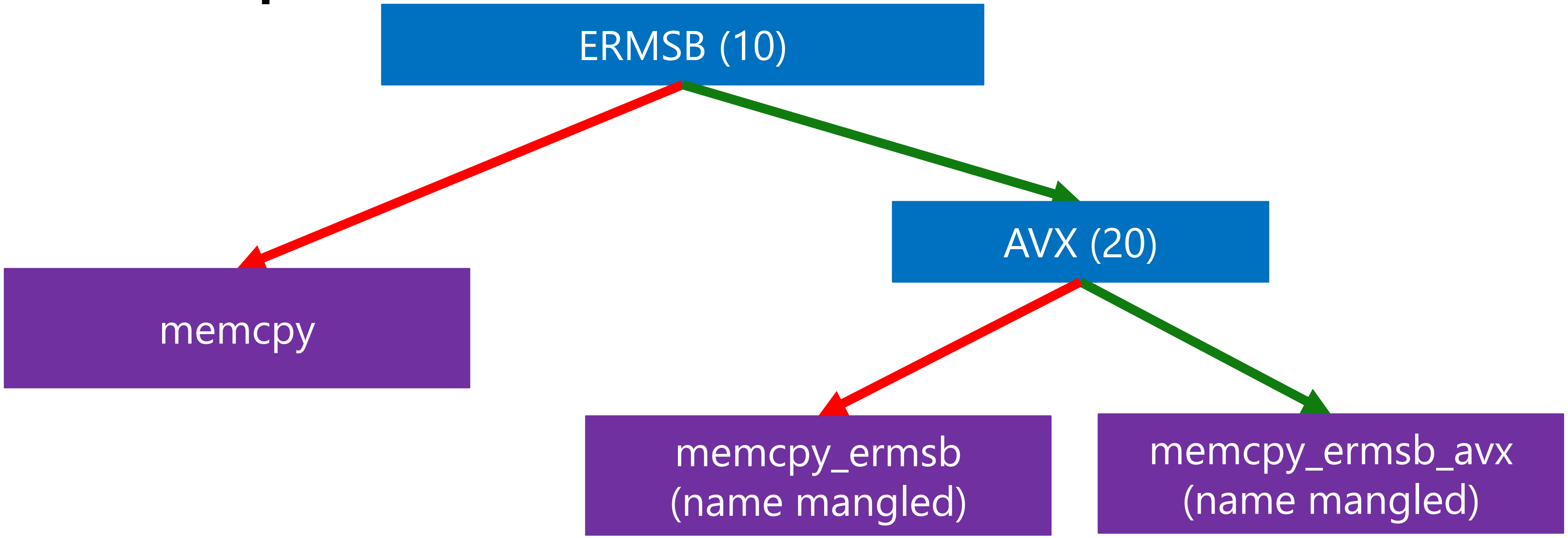
```
[[msvc::dispatch(qualifier: [all(AVX, ERMSB)])]]
void* memcpy(void*, void*, size_t);
```

TU 2:

```
[[msvc::dispatch(qualifier: [all(ERMSB)])]]
void* memcpy(void*, void*, size_t);
```

Assume numerical value as follows: AVX (20) > ERMSB (10)

Qualifier Examples



Order in which capabilities are evaluated based on their numerical value, sorted by linker.

Function names are mangled by compiler (but friendly names displayed here for clarity).

Qualifier Issues

Easy to use for simple cases, but automatic linker ordering can get confusing FAST!

Imagine a case where you have 200 implementations of memcpy.

Ordered Map

Provides ordered evaluation criteria. Conceptually similar to having a bunch of “if” statements.

First set of capabilities that is fully met (in the order defined by the developer) gets selected.

Ordered Map

```
[[msvc::dispatch(map:
    [ all(ERMSB, AVX2) -> memcpy_ermsb_avx2,
      all(ERMSB, AVX) -> memcpy_ermsb_avx,
      ERMSB -> memcpy_ermsb,
      AVX -> memcpy_avx])]]
void* memcpy(void*, void*, size_t);
```

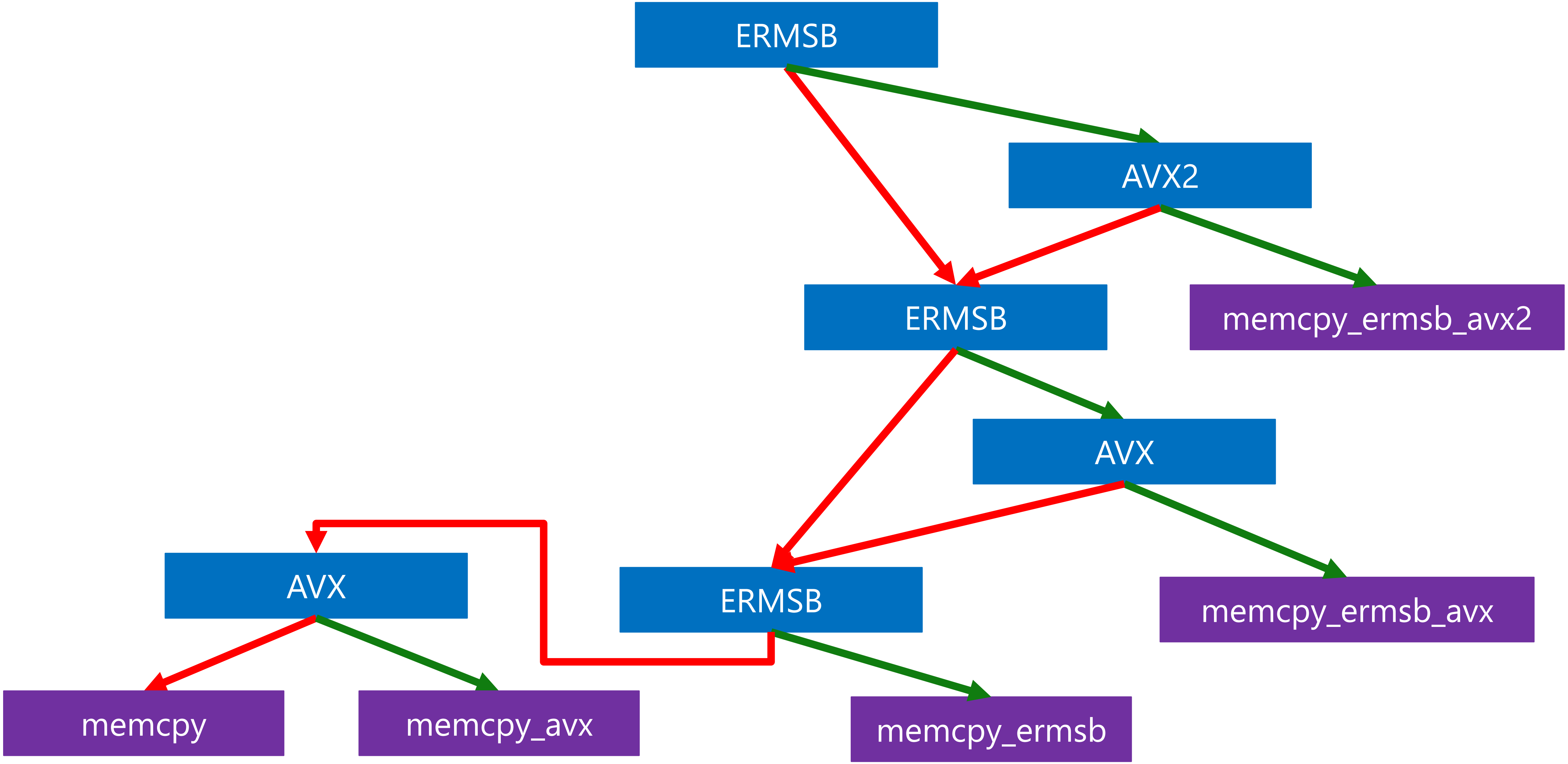
If both ERMSB and AVX2 are supported, use memcpy_ermsb_avx2

If both ERMSB and AVX are supported, use memcpy_ermsb_avx

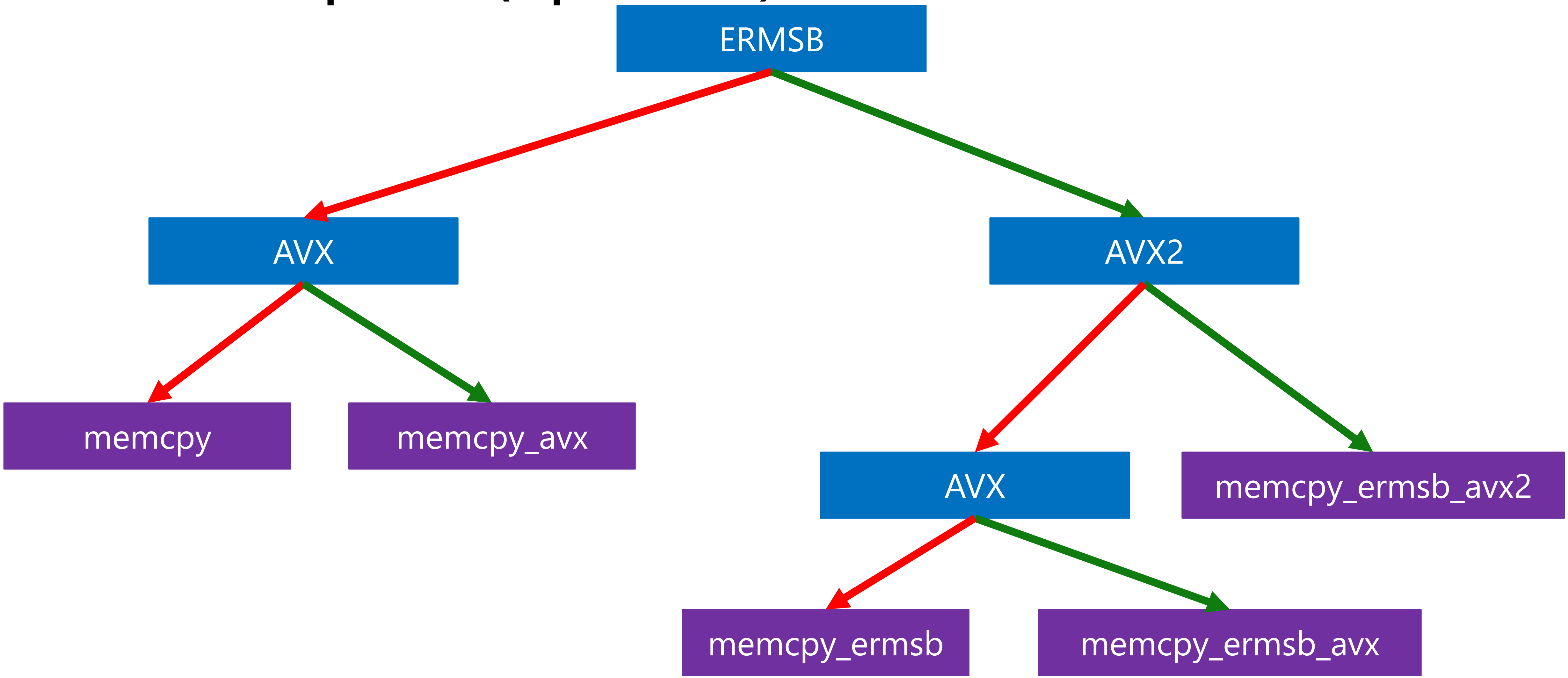
If ERMSB is supported, use memcpy_ermsb

If AVX is supported, use memcpy_avx

Ordered Map BDD (Unoptimized)



Ordered Map BDD (Optimized)



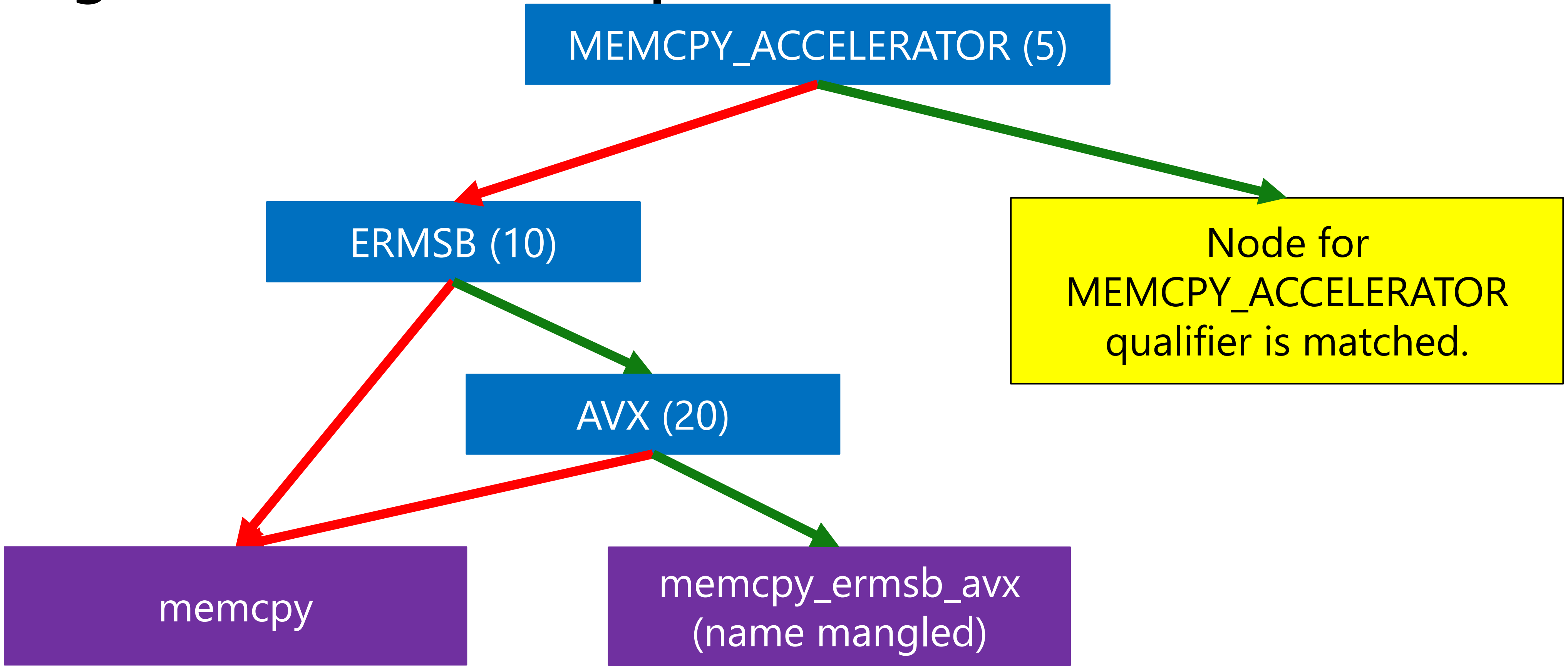
BDD can be compressed but evaluation must follow developer specified order.

Combining Qualifier and Ordered Map

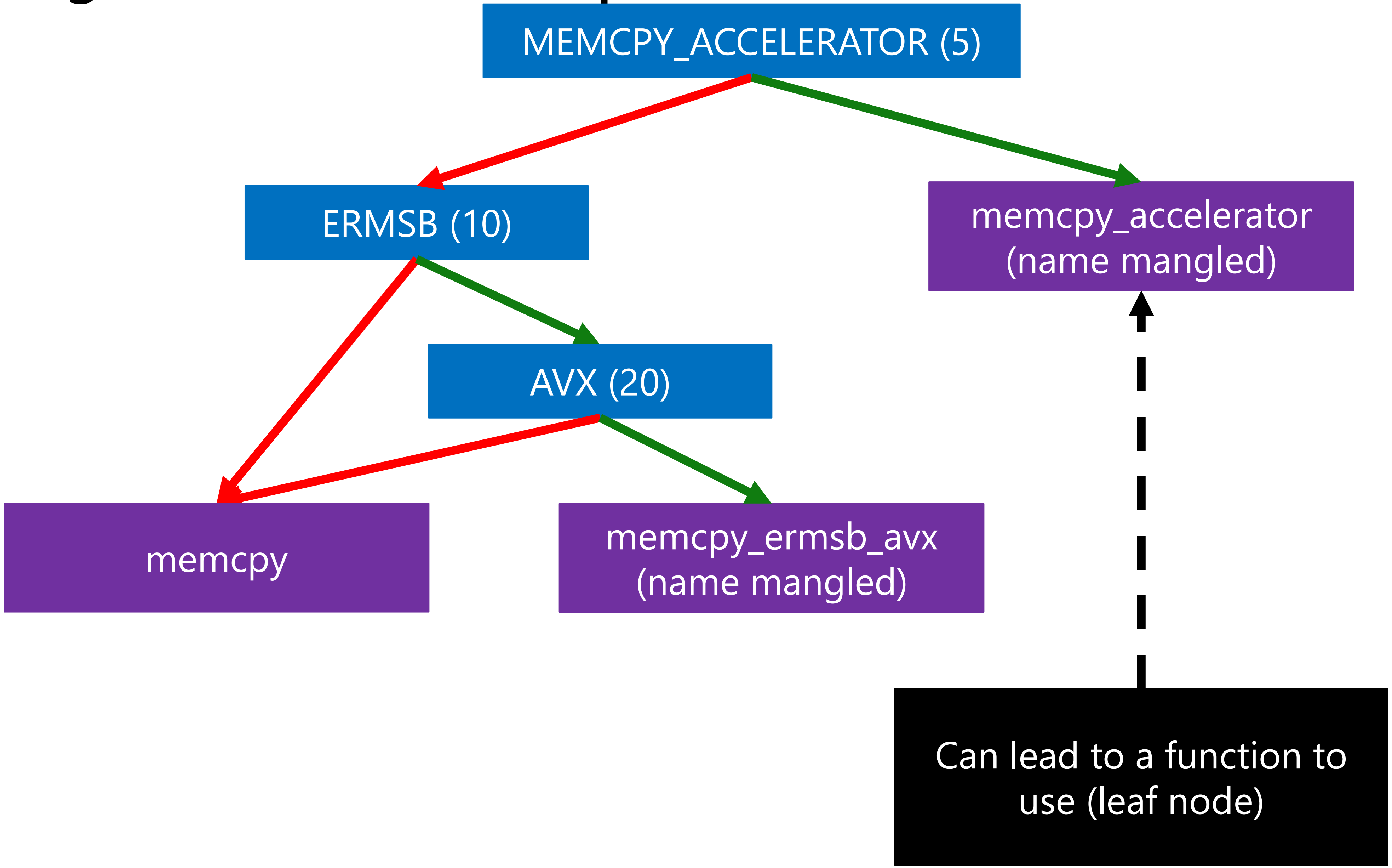
When a qualifier is matched, it leads to either:

- A function to use
- An ordered map

Original Qualifier Example



Original Qualifier Example



Combining Qualifier and Map

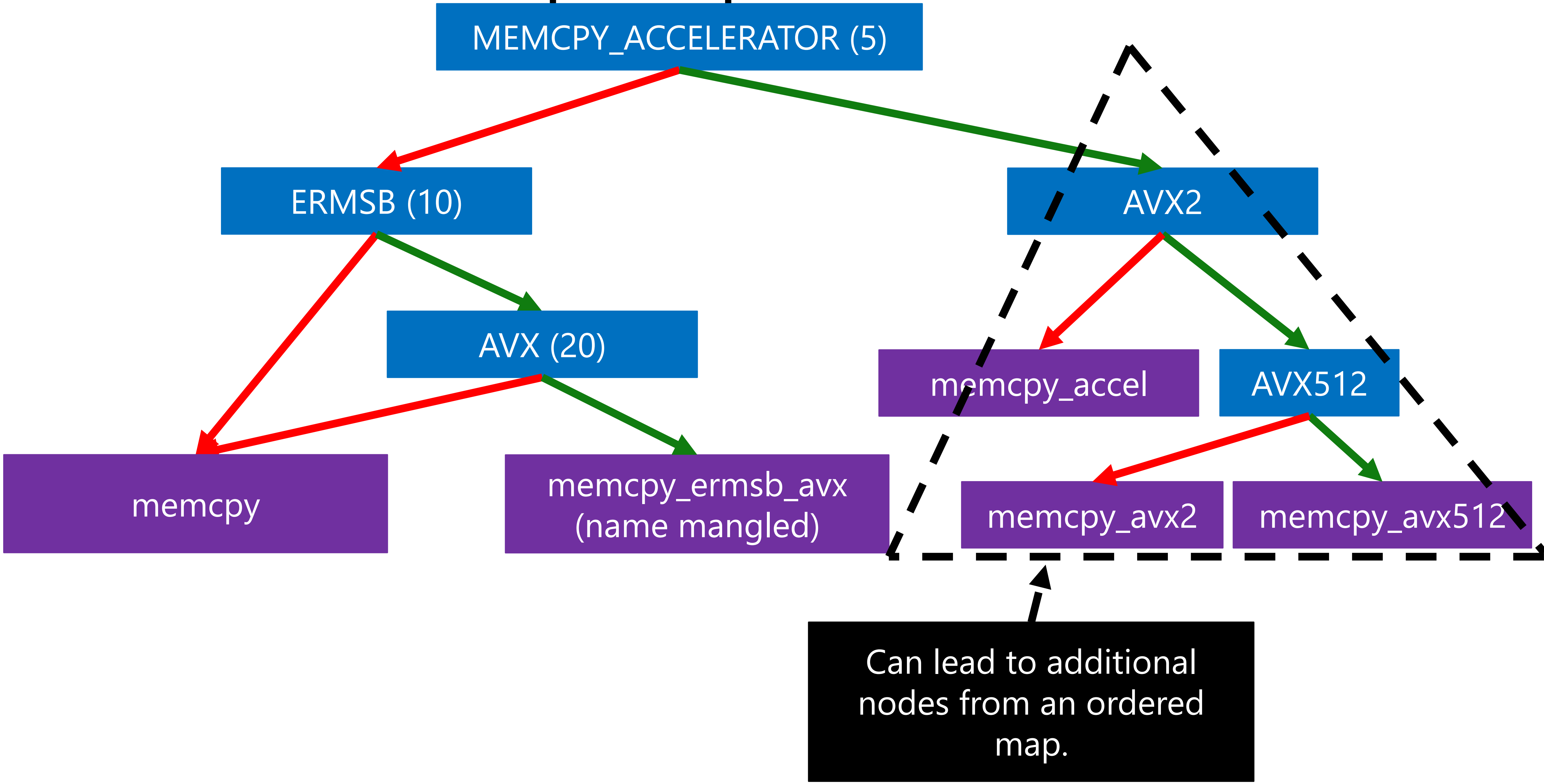
TU 1:

```
[[msvc::dispatch(qualifier: [all(AVX, ERMSB)])]]
void* memcpy(void*, void*, size_t);
```

TU 2:

```
[[msvc::dispatch(qualifier: [all(MEMCPY_ACCELERATOR)],
            map: [all(AVX2, AVX512) -> memcpy_avx512,
                  AVX2 -> memcpy_avx2],
            default: memcpy_accelerator)]]
void* memcpy(void*, void*, size_t);
```

Qualifier + Ordered Map Example



Benefits

Allows defining structured, ordered metadata to define to the OS how to select the most optimal functions.

Metadata can be defined in a single TU or in multiple TU’s.

Supports assembly as well as C/C++ implementations.

4. OS Support

Operating System Loader Support

Boot Time

Detect & cache the presence of all possible capabilities.

Binary Load Time

Parse the LTFS metadata and determine the "most optimal" function based on supported capabilities.

Performs "fixups" (binary modifications) so all direct jumps/calls to the target function now point at the most-optimal candidate function.

Toolchain Details

Compiler

Usage

cl.exe /experimental:loadTimeSelection tu1.cpp

Symbol table

For every dispatch-attributed function, fnname, emit a new symbol fnname_$ltfs$

New section (.ltfsmap)

For every TU that contains at least one dispatch-attributed function, store the dispatch metadata in .ltfsmap section.

LTFSMAP section contains everything required to construct BDDs at link time.

Compiler

Disables inlining of dispatch-attributed functions

LTFS can only apply to direct calls or jumps. If compiler inlines a function, we cannot take advantage of this feature

Disables bottom-up inter-procedural register analysis

Normally, codegen of the callers is dependent on codegen of their callees. Callers can make use of volatile registers which are not used by its callees without saving/restoring them across calls.

In LTFS, to preserve correctness, we disable this optimization where dispatch-attributed is called.

More LTFS syntax

Combination of capabilities

```
[[msvc::dispatch(map:
            [any(AVX, AVX2) -> foo_avx,
             all(ERMSB, POPCNT) -> foo_ermsb_pop,
             none(SSE4_1, SSE4_2) -> foo_simple])]]
int foo(int a) {
    return 42;
}
```

Constexpr support

```
[[msvc::dispatch(map: [any(AVX2, AVX512) -> foo_avx,
                       none(ERMSB, ACCEL) -> foo_simple])]]
constexpr int foo(int a) {
    if (std::is_constant_evaluated()) {
        // instructions to evaluate at compile-time
    } else {
        // instruction to evaluate at runtime
    }
}
```

Life of a dispatch-attributed function, foo

tu3.cpp

```
#include "cap.h"

int foo_avx512(int);
int foo_avx2(int);

[[msvc::dispatch(map: [
         AVX512 -> foo_avx512,
         AVX2 -> foo_avx2
         ])]]
int foo(int) {  return 3; }

int main() { return foo(3); }
```

cl.exe /c /experimental:loadTimeSelection tu3.cpp

```
COFF SYMBOL TABLE
000 01057C57 ABS    notype       Static       | @comp.id
001 80010190 ABS    notype       Static       | @feat.00
002 00000002 ABS    notype       Static       | @vol.md
003 00000000 SECT1  notype       Static       | .drectve
    Section length   2F, #relocs    0, #linenums    0, checksum        0
005 00000000 SECT2  notype       Static       | .debug$S
    Section length   6C, #relocs    0, #linenums    0, checksum        0
007 00000000 SECT3  notype       Static       | .text$mn
    Section length   43, #relocs    1, #linenums    0, checksum 5A7C20DF
009 00000000 SECT3  notype ()    External     | ?foo_avx512@@YAHH@Z (int __cdecl foo_avx512(int))
00A 00000010 SECT3  notype ()    External     | ?foo_avx2@@YAHH@Z (int __cdecl foo_avx2(int))
00B 00000000 ABS    notype       External     | ?foo@@YAHH@Z_$ltfs$ (?foo@@YAHH@Z_$ltfs$)
00C 00000020 SECT3  notype ()    External     | ?foo@@YAHH@Z (int __cdecl foo(int))
00D 00000030 SECT3  notype ()    External     | main
00E 00000030 SECT3  notype       Label        | $LN3
00F 00000000 SECT4  notype       Static       | .xdata
    Section length    8, #relocs    0, #linenums    0, checksum  FC539D1
011 00000000 SECT4  notype       Static       | $unwind$main
012 00000000 SECT5  notype       Static       | .pdata
    Section length    C, #relocs    3, #linenums    0, checksum F23436C8
014 00000000 SECT5  notype       Static       | $pdata$main
015 00000000 SECT6  notype       Static       | .ltfsmap
    Section length   63, #relocs    0, #linenums    0, checksum        0
017 00000000 SECT7  notype       Static       | .chks64
    Section length   38, #relocs    0, #linenums    0, checksum        0

String Table Size = 0x64 bytes

  Summary

          38 .chks64
          6C .debug$S
          2F .drectve
          63 .ltfsmap
           C .pdata
          43 .text$mn
           8 .xdata
```

link.exe /dump /symbols tu3.obj

cl.exe /c /experimental:loadTimeSelection tu3.cpp

```
Dump of file tu3.obj

File Type: COFF OBJECT

SECTION HEADER #6
.ltfsmap name
       0 physical address
       0 virtual address
      63 size of raw data
     246 file pointer to raw data (00000246 to 000002A8)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only

RAW DATA #6
  00000000: 4D 53 56 43 44 49 53 50 41 54 43 48 01 00 00 00  MSVCDISPATCH....
  00000010: 01 00 00 00 3F 66 6F 6F 40 40 59 41 48 48 40 5A  ....?foo@@YAHH@Z
  00000020: 00 02 00 00 00 04 00 00 00 01 00 00 00 05 00 00  ................
  00000030: 00 3F 66 6F 6F 5F 61 76 78 35 31 32 40 40 59 41  .?foo_avx512@@YA
  00000040: 48 48 40 5A 00 04 00 00 00 01 00 00 00 04 00 00  HH@Z............
  00000050: 00 3F 66 6F 6F 5F 61 76 78 32 40 40 59 41 48 48  .?foo_avx2@@YAHH
  00000060: 40 5A 00                                          @Z.
```

link.exe /dump /rawdata /section:.ltfsmap tu3.obj

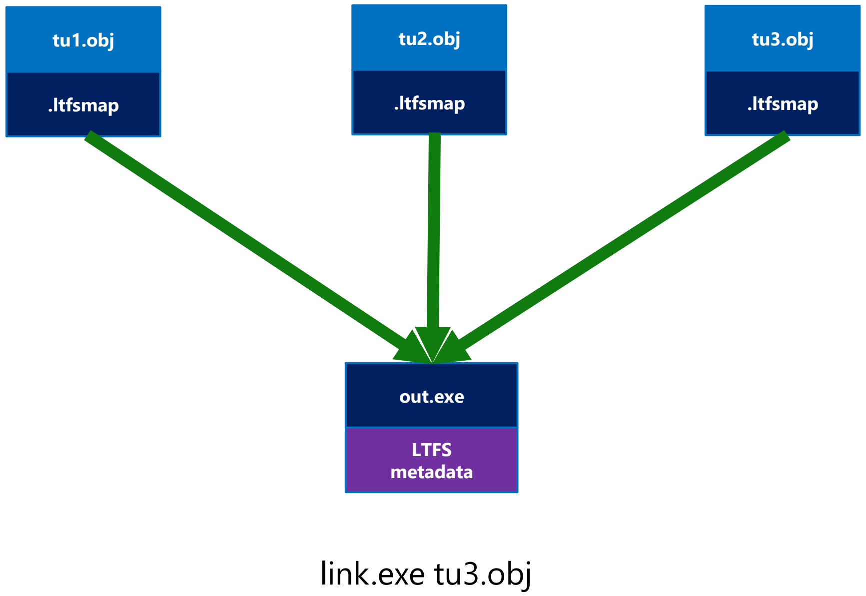
Linker

Error if any function is dispatch-attributed in one TU but not in other.

Collects all the callsites for every dispatch-attributed function.

Final BDD is constructed from .ltfsmap section followed by its topological sort.

Generates thunks.



Supporting Indirect Calls

The linker modifies any code taking the address of the target function (foo) to instead take the address of the generated thunk (foo_thunk)

Pointer comparisons at runtime work irrespective of the function that gets finally selected by the loader.

For DLL exports, thunk function is the one that gets exported.

Thunks for direct calls

By default, direct calls to foo are also patched to call foo_thunk

ARM64 specifics

ARM64 direct call/jump uses BL/B instruction that can support jumps only within the range of +-128MB.

Compiler inserts branch islands when target is farther than this.

Since target in LTFS is unknown until load-time, we have to take this into account. The branch island is now inserted based on the RVA of the candidate which is farthest to the callsite.

Final binary with thunks

link.exe tu3.obj

```
Symbol VA: 0000000000000007 IMAGE_DYNAMIC_RELOCATION_FUNCTION_OVERRIDE
Total function overrides size: 24 bytes
BDD data: 30 bytes

Function Override (1)

    Original RVA:    00001020 ?foo@@YAHH@Z (int __cdecl foo(int))
    BDD Offset:             0
    RVA array size:         8
    Reloc size:             C

    RVAs:

        [00000000] 00001000 ?foo_avx512@@YAHH@Z (int __cdecl foo_avx512(int))
        [00000001] 00001010 ?foo_avx2@@YAHH@Z (int __cdecl foo_avx2(int))

    BDD (version: 1, 28 bytes):

        [00000000] 0002, 0001, 00000005
        [00000001] 0001, 0001, 00000000
        [00000002] 0004, 0003, 00000004
        [00000003] 0003, 0003, 00000001
        [00000004] 0000, 0000, 00000000

    Fixup RVAs:

        [00000000] = page 0005D000 rva 0005D011 type 1
```

link /dump /loadconfig /relocations test.exe

```
main:
  0000000140001030: 48 83 EC 28        sub      rsp,28h
  0000000140001034: B9 03 00 00 00     mov      ecx,3
  0000000140001039: E8 D2 BF 05 00     call     ?foo@@YAHH@Z$thunk$17569504051137690042
  000000014000103E: 48 83 C4 28        add      rsp,28h
  0000000140001042: C3                 ret
  0000000140001043: CC
```

```
?foo@@YAHH@Z$thunk$17569504051137690042:
  000000014005D010: E9 0B 40 FA FF     jmp      ?foo@@YAHH@Z
  000000014005D015: CC                 int      3
  000000014005D016: CC                 int      3
```

link /dump /disasm test.exe

Final binary without thunks

link.exe /ltfs-on-callsites tu3.obj

```
Symbol VA: 0000000000000007 IMAGE_DYNAMIC_RELOCATION_FUNCTION_OVERRIDE
Total function overrides size: 30 bytes
BDD data: 30 bytes

Function Override (1)

    Original RVA:    00001020 ?foo@@YAHH@Z (int __cdecl foo(int))
    BDD Offset:             0
    RVA array size:         8
    Reloc size:            18

    RVAs:

        [00000000] 00001000 ?foo_avx512@@YAHH@Z (int __cdecl foo_avx512(int))
        [00000001] 00001010 ?foo_avx2@@YAHH@Z (int __cdecl foo_avx2(int))

    BDD (version: 1, 28 bytes):

        [00000000] 0002, 0001, 00000005
        [00000001] 0001, 0001, 00000000
        [00000002] 0004, 0003, 00000004
        [00000003] 0003, 0003, 00000001
        [00000004] 0000, 0000, 00000000

    Fixup RVAs:

        [00000000] = page 00001000 rva 0000103A type 1
        [00000001] = page 0005D000 rva 0005D011 type 1
```

link /dump /loadconfig /relocations test.exe

```
main:
  0000000140001030: 48 83 EC 28        sub         rsp,28h
  0000000140001034: B9 03 00 00 00     mov         ecx,3
  0000000140001039: E8 E2 FF FF FF     call        ?foo@@YAHH@Z
  000000014000103E: 48 83 C4 28        add         rsp,28h
  0000000140001042: C3                 ret
  0000000140001043: CC
```

link /dump /disasm test.exe

Relocation types

```
Symbol VA: 0000000000000007 IMAGE_DYNAMIC_RELOCATION_FUNCTION_OVERRIDE
Total function overrides size: 24 bytes
BDD data: 30 bytes

Function Override (1)

    Original RVA:    00001020 ?foo@@YAHH@Z (int __cdecl foo(int))
    BDD Offset:             0
    RVA array size:         8
    Reloc size:             C

    RVAs:

        [00000000] 00001000 ?foo_avx512@@YAHH@Z (int __cdecl foo_avx512(int))
        [00000001] 00001010 ?foo_avx2@@YAHH@Z (int __cdecl foo_avx2(int))

    BDD (version: 1, 28 bytes):

        [00000000] 0002, 0001, 00000005
        [00000001] 0001, 0001, 00000000
        [00000002] 0004, 0003, 00000004
        [00000003] 0003, 0003, 00000001
        [00000004] 0000, 0000, 00000000

    Fixup RVAs:

        [00000000] = page 0005D000 rva 0005D011 type 1
```

link /dump /loadconfig /relocations test.exe

Type 1 = AMD64 call/jmp instructions

Type 2 = ARM64 BL/B instructions

Type 3 = ARM64 ADRP/ADD/BR instruction seq.

```
?foo@@YAHH@Z$thunk$17569504051137690042:
  000000014005D010: E9 0B 40 FA FF      jmp          ?foo@@YAHH@Z
  000000014005D015: [illegible]                 int          3
  000000014005D0[illegible]: CC                 int          3
```

link /dump /disasm test.exe

Relocation types

```
Function Override (5)

    Original RVA:     00001EA0 ?foo@@YAHH@Z (int __cdecl foo(int))
    BDD Offset:             C0
    RVA array size:          8
    Reloc size:              C

    RVAs:

        [00000000] 00001E70 ?foo_avx512@@YAHH@Z (int __cdecl foo_avx512(int))
        [00000001] 00001E88 ?foo_avx2@@YAHH@Z (int __cdecl foo_avx2(int))

    BDD (version: 1, 28 bytes):

        [00000000] 0002, 0001, 00000005
        [00000001] 0001, 0001, 00000000
        [00000002] 0004, 0003, 00000004
        [00000003] 0003, 0003, 00000001
        [00000004] 0000, 0000, 00000000

    Fixup RVAs:

        [00000000] = page 00070000 rva 00070050 type 3
```

link /dump /loadconfig /relocations test.exe

Type 1 = AMD64 call/jmp instructions
Type 2 = ARM64 BL/B instructions
Type 3 = ARM64 ADRP/ADD/BR instruction seq.

```
?foo@@YAHH@Z$thunk$17569504051137690042:
  0000000140070050: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070054: 913A8210  add         xip0,xip0,#0xEA0
  0000000140070058: D61F0200  br          xip0
  000000014007005C: 00000000
```

link /dump /disasm test.exe

Thunk page

```
  0000000140070000: 00000000
  0000000140070004: 00000000
  0000000140070008: 00000000
  000000014007000C: 00000000
?foo4@@YAHH@Z$thunk$8196430937487412666:
  0000000140070010: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070014: 913C0210  add         xip0,xip0,#0xF00
  0000000140070018: D61F0200  br          xip0
  000000014007001C: 00000000
?foo2@@YAHH@Z$thunk$1633091703813367566:
  0000000140070020: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070024: 913B4210  add         xip0,xip0,#0xED0
  0000000140070028: D61F0200  br          xip0
  000000014007002C: 00000000
?foo1@@YAHH@Z$thunk$11956391076113323437:
  0000000140070030: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070034: 913AE210  add         xip0,xip0,#0xEB8
  0000000140070038: D61F0200  br          xip0
  000000014007003C: 00000000
?foo3@@YAHH@Z$thunk$5205080467609598539:
  0000000140070040: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070044: 913BA210  add         xip0,xip0,#0xEE8
  0000000140070048: D61F0200  br          xip0
  000000014007004C: 00000000
?foo@@YAHH@Z$thunk$17569504051137690042:
  0000000140070050: B0FFFC90  adrp        xip0,_guard_dispatch_icall_nop
  0000000140070054: 913A8210  add         xip0,xip0,#0xEA0
  0000000140070058: D61F0200  br          xip0
  000000014007005C: 00000000
```

Disassembly

```
Dump of file tu3.exe

File Type: EXECUTABLE IMAGE

  Summary

        3000 .data
        1000 .ltfsmap
        5000 .pdata
       12000 .rdata
        1000 .reloc
       5C000 .text
        1000 _RDATA
        1000 fothk
```

Section summary

OS Details

Fixup Optimization

Windows doesn’t perform fixups at binary load time and doesn’t write “paged out” code pages to disk.

Kernel caches the information needed to perform fixups.

When a page is accessed (and not currently mapped) we read it from the binary on-disk and apply fixups.

Clustered paging is always fast since the binary's pages are contiguous on-disk.

Why is a Thunk Necessary For Direct Calls/Jumps?

Performing lots of fixups is expensive!

Don’t want to re-write half of a page with fixups prior to it being used.
Thunk minimizes the number of fixups that need to be applied.

Why is a Thunk Necessary For Direct Calls/Jumps?

- In Hyper-V containers (also WDAG, Windows Sandbox, etc.), code pages are shared between host and VM.
  - Saves hundreds of MB of physical memory per container.

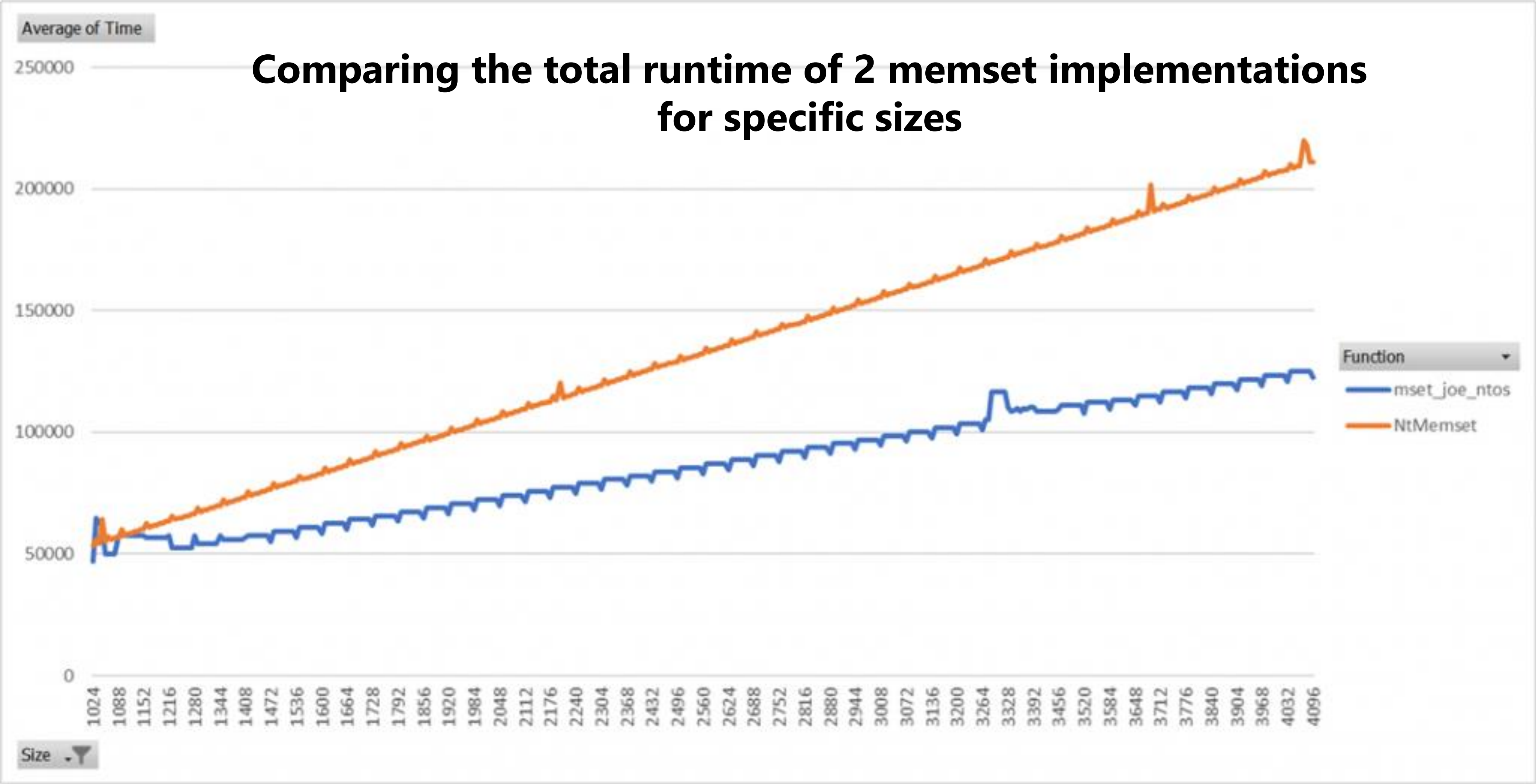
- Code pages that have fixups are NOT shared.
  - Host and container may have different image bases, etc.
  - In practice not many code pages have fixups since AMD64/ARM64 code is position independent.

- LTFS could introduce fixups on a LOT of pages, destroy all code page sharing for containers.
  - Thunks mean only the thunk pages cannot be shared, MUCH smaller overhead.
  - Thunks are clustered together on a small set of pages (or single page).

Demo & Performance Info

https://www.youtube.com/watch?v=tH4rm2eg-Q0

Why Does This Matter??



Using the most optimal instructions can make a BIG difference on performance!

Performance - Intel

of cycles it takes to make 100,000 calls to an asm function that returns 1 on an Intel Broadwell system:

Direct call: 328613	~3.28 cycles/call
Indirect call (CFG): 657175	~6.57 cycles/call
Indirect call (NO-CFG): 410749	~4.1 cycles/call
Thunk call: 410745	~4.1 cycles/call

In practice CFG overhead MUCH higher (performs a bitmap lookup that isn't as fast in real world).

In practice thunk call should perform better than Indirect Call (NOCFG) when missing prediction info.

Performance - ARM

Big Cores:

Indirect call and LTFS (w/ thunk) performance looks equivalent (micro).

Real-world code, LTFS (w/ thunk) faster (easier on branch predictor).

Little Cores:

LTFS (w/ thunk) is 3 cycles faster than indirect call (micro).

On small memcpy's, 3 cycles is ~15+% of the total memcpy cost.

Both Cores:

LTFS (w/ thunk) equivalent to direct calls.

Real World Use Case - Memcpy

We have a prototype memcpy function for ARM64 that is 20-30% faster than our current memcpy.

Only possible to use on newest version of Windows.

Will use LTFS to switch between the old memcpy and the more performant memcpy.

Zero-overhead switching, and 20-30% perf improvement on compatible versions of Windows.

Real World Use Case - CFG

Control Flow Guard on ARM64 causes non-trivial performance regressions on SPEC2k17.

CFG: For every indirect call, a second indirect call is added to a “check function”.

Replacing the second indirect call with LTFS can reduce overall overhead of CFG by up to ~50%.

This is all preliminary benchmark data, investigations still in progress!

Looking Forward

We have several areas in Windows that will be optimized using this feature soon.

Supported in latest Windows Insider Preview builds, Visual Studio support tentatively planned for 17.5 Preview 1.

We’d love to hear your feedback!

Enjoy the rest of the conference!

Join #visual_studio channel on CppCon Discord
https://aka.ms/cppcon/discord

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
https://aka.ms/cppcon

Our sessions

Monday 12th

- The Imperatives Must Go – Victor Ciura
- What's New in C++ 23 – Sy Brand
- C++ Dependencies Don't Have to Be Painful – Augustin Popa
- How Microsoft Uses C++ to Deliver Office – Zachary Henkel

Tuesday 13th

- High-performance Load-time Implementation Selection – Joe Bialek, Pranav Kant
- C++ MythBusters – Victor Ciura

Wednesday 14th

-memory-safe C++ - Jim Radigan

Thursday 15th

- What's New for You in Visual Studio Code – Marian Luparu, Sinem Akinci
- Overcoming Embedded Development Tooling Challenges – Marc Goodner
- Reproducible Developer Environments – Michael Price

Friday 16th

- What's New in Visual Studio 2022 – Marian Luparu, Sy Brand
- C++ Complexity (Keynote) – Herb Sutter
- GitHub Features Every C++ Developer Should Know – Michael Price