

Leveraging the MPI ABI for Binary-Level Detection of MPI Traits



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Tim Jammer*, Christian Iwainsky and Christian Bischof
Technical University Darmstadt

NHR4
CES

NHR for
Computational
Engineering
Science



Scientific
Computing



TECHNISCHE
UNIVERSITÄT
DARMSTADT

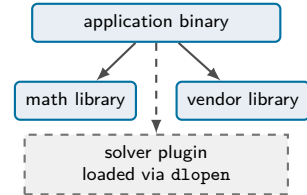
* Work conducted at TU Darmstadt; now at RooflineAI

jammer@roofline.ai



<https://github.com/tudasc/ancor-program-traits>

- Modern applications:
 - ▣ One binary *and*
 - ▣ Set of shared libraries
- Execution environment defines
 - ▣ Which libraries?
 - ▣ Which versions?
- Behaviour is the **combination** of all components
- Properties are often known per component at build time ...
- ...but that knowledge is lost at the binary interface



Case study: MPI matching wildcards



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- `MPI_ANY_SOURCE` / `MPI_ANY_TAG`
- If an application never uses wildcards, **more efficient matching is possible**
- MPI 4.0 lets the programmer say so, via info assertions:

```
mpi_assert_no_any_source, mpi_assert_no_any_tag
```

The catch

Violating an assertion is **undefined behaviour**.

The assertion is only safe if it holds *globally*: application *and* every linked library.

⇒ In practice programmers do not set it, and the optimization opportunity is unused.

- vendor specific closed-source components,
- change of components per run, ...

The MPI ABI changes the picture

MPI Application Binary Interface



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bad news

- A stable MPI ABI makes composition *even more* flexible
- Reasoning about global properties gets harder.

Good news

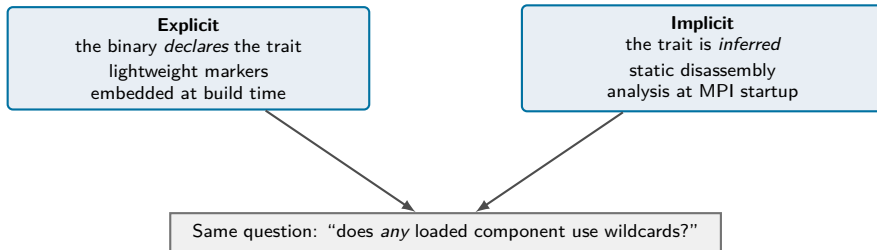
- The ABI fixes what used to be implementation-defined:
 - ▣ concrete values of wildcard constants
 - ▣ calling convention:
which registers hold source and tag

⇒ With a unified ABI, binary-level trait detection becomes implementation independent

Two complementary approaches



TECHNISCHE
UNIVERSITÄT
DARMSTADT



- Both work without source code
- We evaluate overhead, reliability, portability, and extensibility



Explicit Markers

Three candidate mechanisms:

- a dedicated **function symbol** in the dynamic symbol table
- a **string constant** in the read-only data section
- a **custom ELF section** (dropped: not portable across toolchains / build systems)

```
1 // marker function
2 __attribute__((weak)) void marker_function();
3 // prevent removal during dead code elimination:
4 __attribute__((constructor)) static void helper_constructor() {
5     if (marker_function) marker_function(); }
6
7 // alternative: marker string literal
8 __attribute__((weak)) const char marker_str[] = "\xFF_TRAIT_MARKER\0";
9 static const char * volatile keep_marker_str = marker_str;
```

weak: one marker per binary, no matter how many object files.

constructor/ volatile: survive dead code elimination (alternatively: linker flags via mpicc).



- Detecting wildcard usage *within one translation unit* is trivial for a compiler
⇒ it can insert the marker automatically
- No compiler modification needed either: a header macro suffices:

Default-on trick

1. `mpi.h` inserts the “no wildcard” marker **by default** (wildcard usage is rare in practice)
2. In the default case `MPI_ANY_SOURCE` is defined to something *invalid* (e.g. a bitfield of negative length) → **compile error** on use
3. Programs that really need wildcards compile with one preprocessor flag, which restores the real wildcard values and removes the marker

⇒ legacy applications get correct markers with **no build system changes**.

Wildcard-Check at MPI initialization



TECHNISCHE
UNIVERSITÄT
DARMSTADT

MPI_Init

Wildcard-Check at MPI initialization



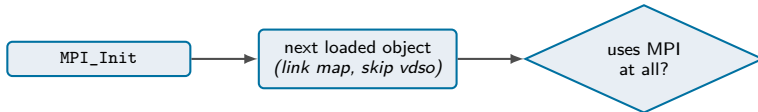
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wildcard-Check at MPI initialization



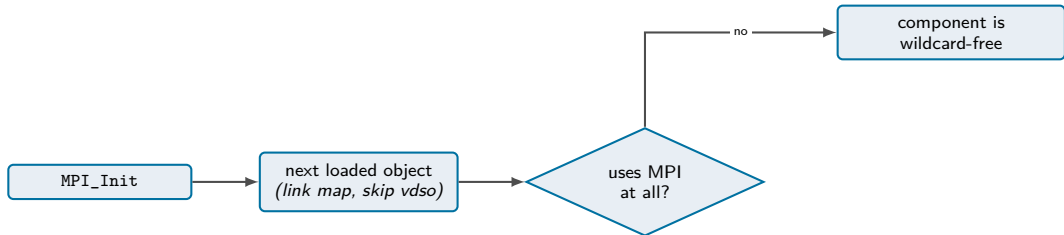
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wildcard-Check at MPI initialization



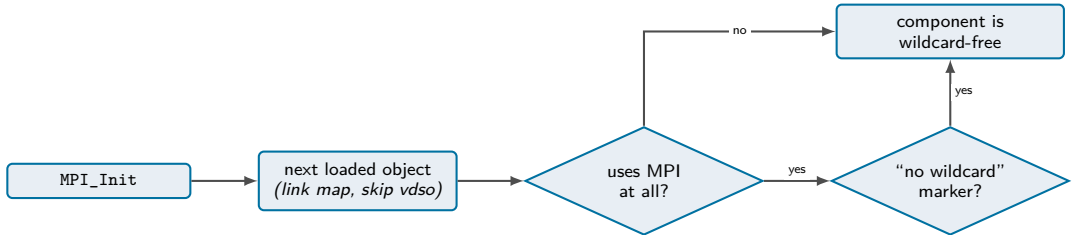
TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wildcard-Check at MPI initialization



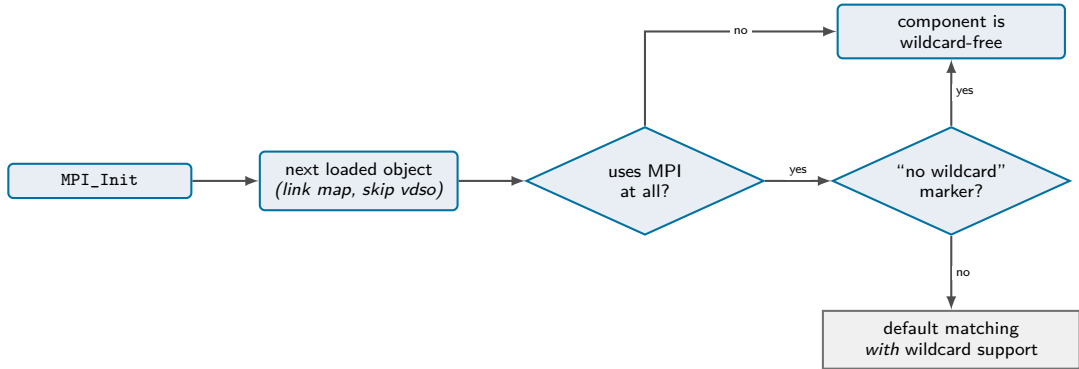
TECHNISCHE
UNIVERSITÄT
DARMSTADT



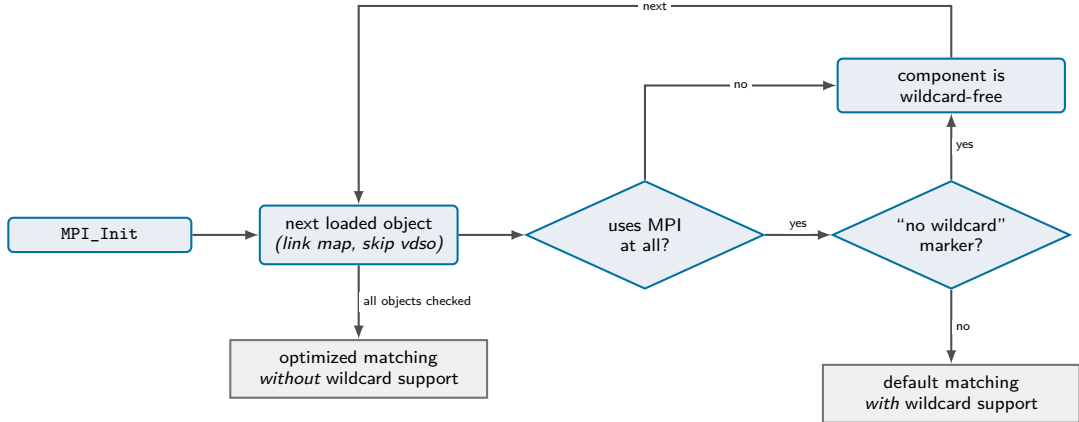
Wildcard-Check at MPI initialization



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Wildcard-Check at MPI initialization



later dlopen may still invalidate this → next slide



- **No marker** $\Rightarrow$ **assume wildcards**: unaware components are automatically safe
- No MPI symbols at all $\Rightarrow$ the component cannot call `MPI_Recv` anyway
- Load order is irrelevant: we check *all* objects, so `LD_PRELOAD` is fine
- Applications may load additional solver module $\rightarrow$ we intercept `dlopen` and re-check
- If the new component violates the trait: fall back to the general matching implementation
 - ▣ this interferes with messages currently “in flight”
 - ▣ loading code that matches to “in flight” messages does not look like good practice to us
 - Use synchronization
 - or an environment variable to force the safe path from program start



Implicit Detection



- Wildcards are **compile-time constants**

⇒ constant propagation usually places them directly at the call site.

1. For every loaded object: does it reference `MPI_Recv` / `MPI_Irecv`?
 - statically linked binaries: the address is directly available
 - under lazy binding the address is not resolved yet
 - but the relocation entry in `.plt` identifies the call sites
2. Disassemble the binary (we use Zydis)
3. ABI-defined calling convention on x86-64: source in `%rcx`, tag in `%r8`
4. Analyse the basic block preceding each call site: can a **negative value** reach those registers?

If yes ⇒ wildcard usage.

Analyzing the disassembly

It is not always directly a `mov -1`



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 int rank;
2 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3 int source = (rank == 0)
4             ? MPI_ANY_SOURCE : 0;
5 MPI_Irecv(buffer, 256, MPI_CHAR,
6           source, 0,
7           MPI_COMM_WORLD, &request);
```

```
1 call MPI_Comm_rank
2 mov 0x14(%rsp), %eax ; eax = rank
3 xor %ecx, %ecx      ; source = 0
4 test %eax, %eax     ; rank == 0 ?
5 sete %cl            ; cl = 0 or 1
6 neg %ecx            ; ecx = -1 or 0
7 call MPI_Recv
```

Rank 0 receives from anyone, all others from rank 0.

- No immediate `-1` anywhere: the wildcard is produced by `sete + neg`
- A lightweight **data-flow analysis** over the preceding instructions catches this
- `test` instead of `cmp` is what the compiler prefers here: it saves encoding space



- We disassemble linearly and keep the last 100 instructions in a ring buffer
 - ▣ constant propagation puts the relevant definition close to the call in practice
- The data-flow analysis **stops** at complex control flow or memory-dependent value propagation
- No evidence of a negative value $\Rightarrow$ we assume *no* wildcard
 - ▣ this is the **optimistic** direction: a spilled-and-reloaded wildcard could be missed
 - ▣ constructible in principle; we did not observe it (see evaluation)
- dlopen is handled exactly as for the explicit markers

Correctness and portability

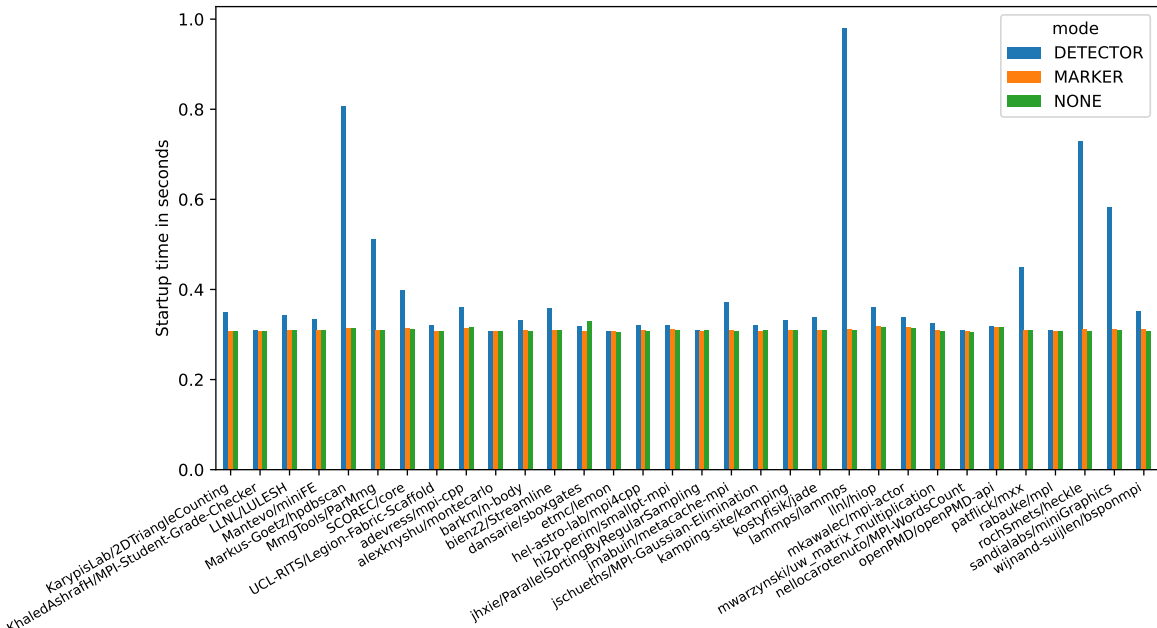
Evaluated with 32 Applications scraped from GitHub



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- **Never** a wrong “wildcard-free” verdict for a receive operation
 - ▣ Validated against source code
 - ▣ Two applications use a wildcard in `MPI_Iprobe`,
 - Proof-of-concept of the disassembly analysis only checks `MPI_Recv` / `MPI_Irecv`.
- Worst case for markers: application is wildcard-free but was built without the marker header
 - ⇒ opportunity missed, nothing broken
- Markers running under an *unaware* MPI implementation are simply ignored
 - ⇒ portability preserved
- Disassembly-based analysis: the compiler propagated wildcard constants to the call site for every usage we found in the 32 applications

Time of MPI initialization





- The cost is paid **once**, at initialization
- MPI startup time is negligible for essentially all production HPC runs
 - if not: MPI is probably the wrong tool for that workload
- Even the disassembly-based approach is unlikely to be noticeable in practice
 - We measured against a fast single-node MPI startup

Marker overhead is comparable to the noise of dynamic linking itself.



	Explicit markers	Disassembly
Startup overhead	low ($\approx 1.8\%$)	up to 128% (variable)
Failure direction	misses the <i>no-wildcard</i> case	may miss a <i>wildcard</i>
Needs compiler / linker cooperation	yes	no
Needs all components rebuilt	yes	no
Works on opaque third-party binaries	if cooperating	yes
Architecture dependence	none	x86-64 decoding
Extends to other traits	naturally	hard

- **Our recommendation: markers.** Compiler cooperation is easy to get through the MPI compiler wrapper
- Markers extend naturally to required threading level, `allow_overtaking`, `exact_length`, ...
 - ▣ difficult to analyze with disassembly



- Both mechanisms work unchanged with `MPI_Session_init`
- Language agnostic: for binary-level multi-language programs are covered by design
- But: not *interpreted* code. Python with `mpi4py`:
 - ▣ calls are dispatched through one generic compiled binding
 - ▣ source / tag are computed from `PyObject`s at runtime (nothing to constant-propagate)
 - ▣ `mpi4py` itself supports wildcards, so it can never carry a “no wildcard” marker
 - ▣ Cython does not fix this, unless the code `cimports` the C-level MPI interface directly

Binary-level trait detection applies where the MPI call is compiled.

- The stable MPI ABI makes **implementation-independent binary-level trait detection** possible for the first time
- Two mechanisms, both without source access:
explicit markers and **disassembly-based inference**
- Overhead is paid once at init:
 $\approx 1.8\%$ (markers), up to 128% (disassembly) of MPI_Init
- Both are **portability-neutral** towards agnostic MPI implementations
- Markers extend beyond wildcards to other MPI assertions and traits
- **Future work:** further traits, compiler-assisted verification



 tudasc/

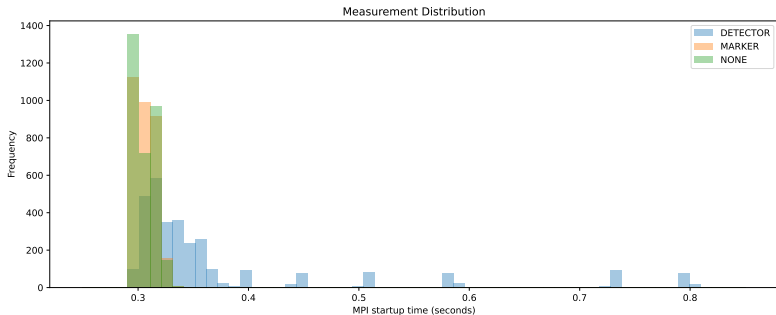
ancor-program-traits

jammer@roofline.ai

Backup: distribution of startup times



TECHNISCHE
UNIVERSITÄT
DARMSTADT



The disassembly-based approach produces the long tail: a few binaries with many instructions and call sites dominate the cost.

Backup: why not a custom ELF section?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Encoding the property in a dedicated section is the “clean” answer
- But its survival depends on toolchain-specific support: compiler attributes, linker script behaviour, and stripping
- Function symbols and string constants are plain, standard C/C++ and survive optimized and stripped builds with a single explicit reference
- Portability across build systems was the deciding criterion



How we validate

- PMPI wrapper checks the trait at `MPI_Init`
- If “wildcard-free” was concluded, it intercepts *all* communication and **aborts** on any actual wildcard use
- Markers injected via a “fake” `mpi.h` on the include path
- All four combinations of aware / unaware build $\times$ run
- ✓ No wrong “wildcard-free” conclusions

Benchmarks and system

- **32 real-world MPI applications** from GitHub
(C/C++, CMake, built unmodified, executable calling `MPI_Init`)
- Intel Xeon Platinum 9242, 768 GB, RHEL 9.6, gcc 11.5, OpenMPI 4.1.8
- 2 ranks, single node, 110 runs, first 10 discarded as warm-up
- Terminate right after init $\rightarrow$ measures *only* startup cost



- Criteria: C/C++ project on GitHub, uses MPI, uses CMake, builds on our system **without modifying build parameters**, produces an executable that calls `MPI_Init`
- Excluded: projects requiring MPI to be explicitly selected at configure time, and library-only projects (we cannot execute a library)
- 32 projects remained, including LULESH, miniFE, lammmps, ParMmg, hiop, openPMD-api, kamping, mxx, mpl, hpdbscan, ...
- Full list and scripts in the repository

Backup: what happens on a marker mismatch?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Marker present, MPI unaware of the mechanism → marker ignored, program behaves normally
- No marker, MPI aware → wildcards assumed, i.e. today's behaviour
- Marker present but wrong (application does use wildcards) → this is the same undefined-behaviour situation as a wrongly set MPI 4.0 assertion; our PMPI wrapper detects and aborts on it during testing
- An MPI_Allreduce is used to agree across ranks, though strictly it is not required — different ranks may use different matching implementations