Eliminate Branches by Melding IR Instructions (Artifact)

Yuze Li
Virginia Tech, Blacksburg, VA, USA

Srinivasan Ramachandra Sharma
Virginia Tech, Blacksburg, VA, USA

Charitha Saumya
Intel Corporation, Santa Clara, CA, USA

Ali R. Butt
Virginia Tech, Blacksburg, VA, USA

Kirshanthan Sundararajah
Virginia Tech, Blacksburg, VA, USA

Abstract

Branch mispredictions cause catastrophic performance penalties in modern processors, leading to performance loss. While hardware predictors and profile-guided techniques exist, data-dependent branches with irregular patterns remain challenging. Traditional if-conversion eliminates branches via software predication but faces limitations on architectures like x86. It often fails on paths containing memory instructions or incurs excessive instruction overhead by fully speculating large branch bodies.

This paper presents Eliminate Branches by Melding IR Instructions (MERIT), a compiler transformation that eliminates branches by aligning and melding similar operations from divergent paths at the IR instruction level. By observing that divergent paths often perform structurally similar operations with different operands, MERIT adapts sequence alignment to discover merging opportunities and employs safe operand-level guarding to ensure semantic correctness without hardware predication. Implemented as an LLVM pass and evaluated on 102 programs from four benchmark suites, MERIT achieves a geometric mean speedup of 1.51× on 24 branch-heavy microbenchmarks (peak 32× on toUpper), reducing branch mispredictions by 48% on average. On realistic workloads – SPECrate 2017 (16 benchmarks), SQLite TPC-H (22 queries), and CPython pyperformance (40 benchmarks) – MERIT with profile-guided function selection achieves ∼1% geometric mean speedup across all three suites, demonstrating consistent improvement without regressions on diverse production workloads.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Source code generation

Keywords and phrases Branch elimination, Compiler transformation, LLVM-IR

Digital Object Identifier 10.4230/DARTS.12.1.1

Funding This work is sponsored in part by the NSF under the grants: CSR-2106634 and CSR-2312785.

Acknowledgements We thank the ECOOP artifact evaluation committee for their thorough review.

Related Article Yuze Li, Srinivasan Ramachandra Sharma, Charitha Saumya, Ali R. Butt, and Kirshanthan Sundararajah, "Eliminate Branches by Melding IR Instructions", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 18:1–18:30, 2026. https://doi.org/10.4230/LIPIcs.ECOOP.2026.18

Related Conference 40th European Conference on Object-Oriented Programming (ECOOP 2026), June 29–July 3, 2026, Brussels, Belgium

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2026 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact supports all experimental claims in the conference paper [3]. Specifically, it enables reproduction of the following results:

1.1 Microbenchmarks (Figures 3 and 5)

24 branch-heavy benchmarks spanning sorting, graph, array, and numeric workloads. The paper claims MERIT achieves up to 32× speedup (toUpper) with a geometric mean of 1.51× compared to hardware branch prediction, and 1.44× for MERIT-O2 (Section 5.2). Early if-conversion achieves only 1.06× geomean.

1.2 SPECrate 2017 (Figures 4 and 6)

16 C/C++ benchmarks from SPEC CPU 2017 [5]. The paper claims MERIT reduces 6.1% branch mispredictions on average, with MERIT-PGO achieving a geomean of 1.01× by selectively excluding harmful functions (Section 5.3). Key results: 505.mcf_r improves 4% under PGO despite 14% regression under forced MERIT; 538.imagick_r recovers from 7% regression to neutral.

1.3 SQLite TPC-H (Figure 7)

22 TPC-H [7] queries on SQLite [6]. The paper claims MERIT-PGO achieves a positive geomean by selectively including only beneficial functions, while forced MERIT shows regressions on complex queries (e.g., 0.56× on Query 11) due to instruction overhead in the VdbeExec interpreter loop (Section 5.4).

1.4 CPython pyperformance (Figure 8)

40 benchmarks from the pyperformance [4] suite on CPython 3.10 [1]. The paper claims MERIT achieves 1.01× geomean speedup with peak improvement of 1.11× on unpickle_list, demonstrating effectiveness even on complex interpreter workloads (Section 5.5).

1.5 Compilation Statistics (Paper Table 1)

Compile time overhead, number of transformation sites, and code size changes for all 14 SPEC benchmarks, SQLite, and CPython. The paper reports that MERIT's compile overhead tracks the number of transformation sites (up to 31.9% on 500.perlbench_r) and that code size changes are small (typically ≤5%). The artifact reproduces these metrics using `-mllvm -print-ifconv-stats` to count SimplifyCFG folds, EarlyIfConv conversions, and MERIT merges per compilation unit.

2 Content

The artifact is packaged as a Docker [2] image containing all source code, benchmarks, pre-built binaries, and evaluation scripts. The artifact package includes:

- **MERIT Compiler** (source + pre-built, LLVM 14 fork)
  Binary: `/opt/merit/bin/clang` and `/opt/merit/bin/clang++`
  Source: `/opt/merit/llvm-project/` (`llvm/` and `clang/` directories)
  The MERIT pass is in `llvm/lib/Transforms/CFMSE/` and `llvm/lib/Transforms/CFMelder/`

- **Microbenchmarks** (source code, C/C++)
  Location: `/opt/merit-ae/microbench/src/`
  24 benchmark families with compilation and evaluation scripts.
- **SPEC CPU 2017** (pre-built binaries + input data)
  Location: `/opt/merit-ae/spec2017/benchmarks/`
  Pre-built binaries for 16 benchmarks × 5 modes, with reference input data. 6 runcpu configuration files in `spec2017/configs/`.
- **SQLite TPC-H** (source code + query files)
  Location: `/opt/merit-ae/sqlite/`
  SQLite source, TPC-H data generator (tpch-dbgen), 22 SQL query files, and build/run scripts.
- **CPython pyperformance** (source code)
  Location: `/opt/merit-ae/cpython/`
  CPython 3.10 source (commit `f610f9eab92`), pyperformance source, setup and evaluation scripts.
- **Evaluation Scripts**
  Location: `/opt/merit-ae/scripts/`
  `run_all.sh`: Master push-button script (`-quick` for smoke test).
  Per-suite scripts: `run_microbench.sh`, `run_ecoop_rate.sh`, `run_ecoop_tpch.sh`, `run_ecoop_pyperformance.sh`.
  Plotting scripts: `plot_microbench.py`, `plot_spec_rate.py`, `plot_tpch.py`, `plot_pyperformance.py`.
- **Expected Results** (CSV files)
  Location: `*/expected/*.csv` in each suite directory.
  Reference results for validation of reproduced data.
- **README**
  Location: `/opt/merit-ae/README.md`
  Comprehensive documentation mapping paper figures to scripts.

3 Getting the artifact

The artifact is available as a pre-built Docker image from:

- **Zenodo**: `https://doi.org/10.5281/zenodo.19598511` (DOI: `10.5281/zenodo.19598511`)
- **Docker Hub**: `https://hub.docker.com/r/lyuze/merit-ae` (tag: `ecoop2026`)

4 Tested platforms

4.1 Hardware Requirements

- x86-64 CPU with hardware performance counters (required for `perf stat`)
- At least 16 GiB RAM (32 GiB recommended for SPEC and CPython)
- At least 30 GiB free disk space
- Single physical core pinned via `numactl` (core 0 by default; configurable via `MERIT_CORE`)

4.2 Software Dependencies

All dependencies are pre-installed in the Docker image (Ubuntu 22.04 base):
build tools (`build-essential`, CMake, Ninja, `gfortran`), performance measurement (`perf` via `linux-tools-generic`, `libpapi-dev`, `numactl`), the Python plotting stack (`matplotlib`, `numpy`, `pandas`, `scipy`), the pre-built MERIT compiler, all benchmark sources, pre-built SPEC binaries

with input data, and CPython 3.10 + pyperformance. For non-Docker use, install the same packages via `apt` on Ubuntu, or the Debian equivalents (e.g. `linux-perf` in place of `linux-tools-generic`); see the README for the full list.

4.3 Docker Requirements

- Docker must be run with `-privileged` flag to access hardware performance counters.
- The image ships with a self-contained `perf` binary (`linux-tools-generic`), so no host `perf` mount is required. `perf_event_open` is a syscall-stable kernel ABI, which makes the bundled binary work on any host distro/kernel once `-privileged` is passed.
- Alternative to `-privileged`: `-cap-add=SYS_PTRACE -cap-add=SYS_ADMIN` with `-security-opt seccomp=unconfined`.

4.4 Tested Configurations

- Ubuntu 22.04 LTS, Intel Xeon Silver 4314 CPU @ 2.40GHz (32 cores), 64 GB RAM
- Debian 13 "Trixie" via the `linux-perf` package (reviewer-confirmed for kick-the-tires)

4.5 Known Issues

- **Docker Desktop (macOS/Windows) and WSL2** run the container inside a hidden Linux VM without PMU passthrough. They silently produce zeroed `perf stat` values and break the branch-miss / IPC figures. Use native Linux. Cloud VMs without hardware counter passthrough have the same issue.
- If `perf stat` inside the container reports `<not supported>` for CPU events, loosen the host's paranoid setting: `sudo sh -c 'echo 1 > /proc/sys/kernel/perf_event_paranoid'`. Level 1 allows CPU and kernel events, which is what MERIT requires.

5 License

The artifact components are available under the following licenses:

- **MERIT compiler** (LLVM 14 fork): Apache License 2.0 with LLVM Exceptions.
- **Microbenchmarks**: MIT License.
- **SQLite**: Public Domain.
- **CPython 3.10**: Python Software Foundation License.
- **pyperformance**: MIT License.
- **SPEC CPU 2017**: Pre-built binaries and input data are included solely for artifact evaluation purposes. SPEC CPU 2017 is proprietary software; redistribution of source code is prohibited. Users who wish to rebuild from source require a separate SPEC license from `https://www.spec.org/cpu2017/`. The included binaries and inputs are provided under SPEC's fair-use provisions for academic evaluation.

6 MD5 sum of the artifact

44cb95fb50e6bec01ceb6ec6be55058c

7 Size of the artifact

1.7 GB (Docker image)

A Quick Start Guide (Kick-the-Tires)

A.1 Loading the Docker Image

```
docker load -i merit-ae-image.tar.gz
```

A.2 Launching the Container

The container requires `-privileged` for hardware performance counters. The image bundles its own kernel-agnostic `perf` binary, so no host bind-mount is required:

```
docker run --privileged -it \
  -e MERIT_CORE=0 \
  merit-ae
```

This drops you into an interactive shell at `/opt/merit-ae/`, where all scripts and benchmarks are located. The `MERIT_CORE` variable controls which CPU core benchmarks are pinned to (default: 0). Works on any host distro (Ubuntu, Debian, Fedora, etc.) with `-privileged` and a host kernel that permits `perf_event_open` (see Known Issues for `perf_event_paranoid`).

A.3 Compiler Setup

The image includes both a pre-built compiler and the full LLVM source. Users may use either:

```
./scripts/compile_merit.sh              # Use pre-built (instant)
./scripts/compile_merit.sh --from-source # Build from source (~30 min)
```

Both options produce a working compiler at `/opt/merit/bin/clang`. All subsequent scripts use this path automatically.

A.4 Kick-the-Tires (∼2 minutes)

Inside the container, run the quick evaluation:

```
cd /opt/merit-ae
./scripts/run_all.sh --quick
```

This compiles and runs two standalone examples (`to_upper` and `dutch_flag`) with both baseline and MERIT, verifying that the compiler produces correct branchless code with the expected speedups (∼32× on `to_upper`, ∼3× on `dutch_flag`). No benchmark infrastructure or setup is needed.

B Full Evaluation (∼19 hours)

```
cd /opt/merit-ae
./scripts/run_all.sh
```

This sequentially runs all four benchmark suites and generates PDF figures.

C Reproducing Individual Figures

Note: users who already ran `./scripts/run_all.sh` (Appendix B) can skip this section — `run_all.sh` invokes every command below automatically. This section is for users who want to reproduce a single figure or re-run a single suite.

Table 1 maps each paper figure to its reproduction command and output file.

Table 1 Paper figures and reproduction commands.

Figure	Description	Script	Output
3(a)	Microbench speedup	`run_microbench.sh`	`fig3a_speedup.pdf`
3(b)	Microbench branch miss	(same)	`fig3b_branch_miss.pdf`
4(a)	SPEC speedup	`run_ecoop_rate.sh`	`fig4a_spec_speedup.pdf`
4(b)	SPEC branch miss	(same)	`fig4b_spec_branch_miss.pdf`
5(a)	Microbench IPC	`run_microbench.sh`	`fig5a_ipc_inc.pdf`
5(b)	Microbench instr. overhead	(same)	`fig5b_instruction_inc.pdf`
6(a)	SPEC IPC	`plot_spec_rate.py`	`fig6a_spec_ipc_inc.pdf`
6(b)	SPEC instr. overhead	(same)	`fig6b_spec_instr_inc.pdf`
7(a)	SQLite TPC-H speedup	`run_ecoop_tpch.sh`	`fig7a_tpch_speedup.pdf`
7(b)	SQLite TPC-H branch miss	(same)	`fig7b_tpch_branch_miss.pdf`
8	CPython speedup	`run_ecoop_pyperformance.sh`	`fig8_pyperformance.pdf`
Paper Table 1	Compilation statistics	`run_compile_stats.sh`	`compile_stats.csv`

Each suite can be run independently:

```
# Microbenchmarks (~5 hours)
bash microbench/scripts/run_microbench.sh
python3 microbench/scripts/plot_microbench.py \
    microbench/results/microbench_results.csv microbench/figures/

# SPECrate 2017 (~8 hours)
bash spec2017/scripts/setup_spec_runs.sh
bash spec2017/scripts/run_ecoop_rate.sh
python3 spec2017/scripts/plot_spec_rate.py \
    spec2017/results/spec_rate_results.csv spec2017/figures/

# SQLite TPC-H (~1 hour)
bash sqlite/scripts/build_sqlite.sh
bash sqlite/scripts/setup_tpch_db.sh
bash sqlite/scripts/run_ecoop_tpch.sh
python3 sqlite/scripts/plot_tpch.py \
    sqlite/res_ecoop/tpch_results.csv sqlite/figures/

# CPython pyperformance (~4 hours)
bash cpython/scripts/setup_cpython.sh cpython/work
bash cpython/scripts/run_o2_baseline.sh        # O2 baseline first
bash cpython/scripts/run_ecoop_pyperformance.sh # then 3-mode run + CSV
python3 cpython/scripts/plot_pyperformance.py \
    cpython/work/res_ecoop/pyperformance_results.csv cpython/figures/

# Compilation statistics -- SQLite + Python rows of Table 1 (~5 min)
bash scripts/run_compile_stats.sh --sqlite-only
bash scripts/run_compile_stats.sh --cpython-only
# The 14 SPEC rows are pre-recorded by the authors (produced by the
# same scripts, on the same compiler shipped in this image) at:
#   spec2017/expected/compile_stats.csv
# Users with a SPEC CPU 2017 license can re-run via
#   SPEC_DIR=/path/to/cpu2017 bash scripts/run_compile_stats.sh --spec-only
```

D Reduced Evaluation

For time-constrained evaluation, we provide a reduced configuration:

- **Microbenchmarks**: Use `-quick` flag (3 benchmarks, 1 run each, ∼5 min).
- **SPEC**: Run a single benchmark: `bash run_ecoop_rate.sh 557.xz_r` (∼20 min).
- **SQLite**: Run a single query: `bash run_ecoop_tpch.sh 1` (∼10 min).
- **CPython**: Run a single benchmark: `bash run_ecoop_pyperformance.sh unpickle_list` (∼5 min).

Total for reduced evaluation: ∼40 minutes.

E Cleaning Up

To remove all generated results, figures, and CSV files (keeping builds intact):

```
./scripts/clean_all.sh
```

To also remove SQLite/CPython builds and start completely fresh:

```
./scripts/clean_all.sh --builds
```

F Try MERIT for Yourself

The `quick_start/` directory contains two standalone C programs that demonstrate MERIT without any benchmark infrastructure. Inside the container:

```
cd /opt/merit-ae/quick_start
NOLLC="-Xclang -backend-opt-level -Xclang 0"

# Compile and run baseline vs MERIT
clang -O2 -fno-unroll-loops $NOLLC to_upper.c \
    -o baseline -mllvm -enable-merit=0
clang -O2 -fno-unroll-loops $NOLLC to_upper.c \
    -o merit -mllvm -force-merit=1 -mllvm -merit-unsafe-dci=0

./baseline   # ~6400 ms
./merit      # ~195 ms  (32x faster)
```

The `to_upper` loop has a data-dependent branch mispredicted ∼50% of the time. MERIT eliminates the branch by aligning and melding the two paths into a branchless `select`-based sequence at the IR level, removing the misprediction penalty entirely. A second example (`dutch_flag.c`) demonstrates 3-way classification with ∼3× speedup. See `quick_start/README.md` for details.

G Compilation Flags Reference

All benchmarks use `-O2 -fno-unroll-loops` as the base optimization level. The MERIT-specific flags are:

Flag	Description
`-mllvm -enable-merit=0`	Disable MERIT (baseline)
`-mllvm -force-merit=1`	Force MERIT on all eligible branches
`-mllvm -merit-unsafe-dci=0`	Enable safe DCI (required with `force-merit`)
`-mllvm -x86-early-ifcvt`	Enable LLVM's early if-conversion (comparison baseline)
`-Xclang -backend-opt-level -Xclang 0`	Disable backend optimization (isolates MERIT's IR-level effect)
`-mllvm -include-function-names=...`	Apply MERIT only to listed functions (PGO mode)
`-mllvm -exclude-function-names=...`	Skip MERIT for listed functions (PGO mode)

H Profile-Guided Function Selection (PGO)

The MERIT-PGO results in the paper use per-benchmark function include/exclude lists to selectively apply MERIT only to functions where it helps. The artifact includes a script that automates this selection process:

```
bash scripts/generate_pgo_funclist.sh \
  --baseline <baseline_binary> \
  --merit <forced_merit_binary> \
  --args "<benchmark arguments>" \
  --rundir <directory_with_input_files> \
  --workdir <output_directory>
```

The script performs three steps:

1. **Profile both binaries** using `perf record` (cycle-level sampling) on the same workload.
2. **Compute per-function deltas** using `perf diff`, which reports how much each function's cycle share changed between baseline and forced-MERIT.
3. **Classify functions** based on the delta:
   - *Regressed* (delta $> +0.5\%$): MERIT increases cycles in this function — exclude it.
   - *Improved* (delta $< -0.5\%$): MERIT decreases cycles — include it.
   - *Neutral*: no significant change.

   The script outputs whichever list is shorter (include or exclude) as the recommended compiler flag.

For example, on 505.mcf_r the script identifies `primal_bea_mpp` as the sole regressed function (+14% cycles) and recommends:

```
-mllvm -exclude-function-names="primal_bea_mpp"
```

This matches the PGO configuration used in the paper. To save users' time, we have already provided the pre-computed function include/exclude lists for all benchmarks — they are embedded in the SPEC configuration file (`clang-O2-merit-pgo-nollc.cfg`) and the SQLite build script. Users can directly use these lists to reproduce the MERIT-PGO performance numbers without re-running the profiling step. The `generate_pgo_funclist.sh` script is provided for transparency and for users who wish to verify or regenerate the lists independently.

References

1 CPython: The reference implementation of Python. Commit f610f9eab92, branch 3.10. URL: `https://github.com/python/cpython`.

2 Docker: Empowering app development for developers. Accessed: 2026-03-30. URL: `https://www.docker.com/`.

3 Yuze Li, Srinivasan Ramachandra Sharma, Charitha Saumya, Ali R. Butt, and Kirshanthan Sundararajah. Eliminate branches by melding IR instructions. In *40th European Conference on Object-Oriented Programming (ECOOP 2026)*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für In-

formatik, 2026. doi:10.4230/LIPIcs.ECOOP.2026.18.

4 pyperformance: Python performance benchmark suite. Version 1.11.0. URL: https://github.com/python/pyperformance.

5 SPEC CPU 2017. Standard Performance Evaluation Corporation. URL: https://www.spec.org/cpu2017/.

6 SQLite. Public domain database engine. URL: https://www.sqlite.org/.

7 TPC-H: Decision support benchmark. Transaction Processing Performance Council. URL: https://www.tpc.org/tpch/.