Notes on Runtime Reoptimization

(A work in Progress…)

George V. Neville-Neil

AsiaBSDcon 2026, March 22nd, Taipei, Taiwan

What is a keynote?

- A keynote talk is the **main, headline presentation** at a conference, seminar, or event, designed to set the underlying theme, tone, and agenda.
- Typically delivered by a **prominent expert** or **authority**, it inspires and connects the audience to the core message or "big picture" of the gathering.

None of the above!

You did what?! Why?!

Yale University

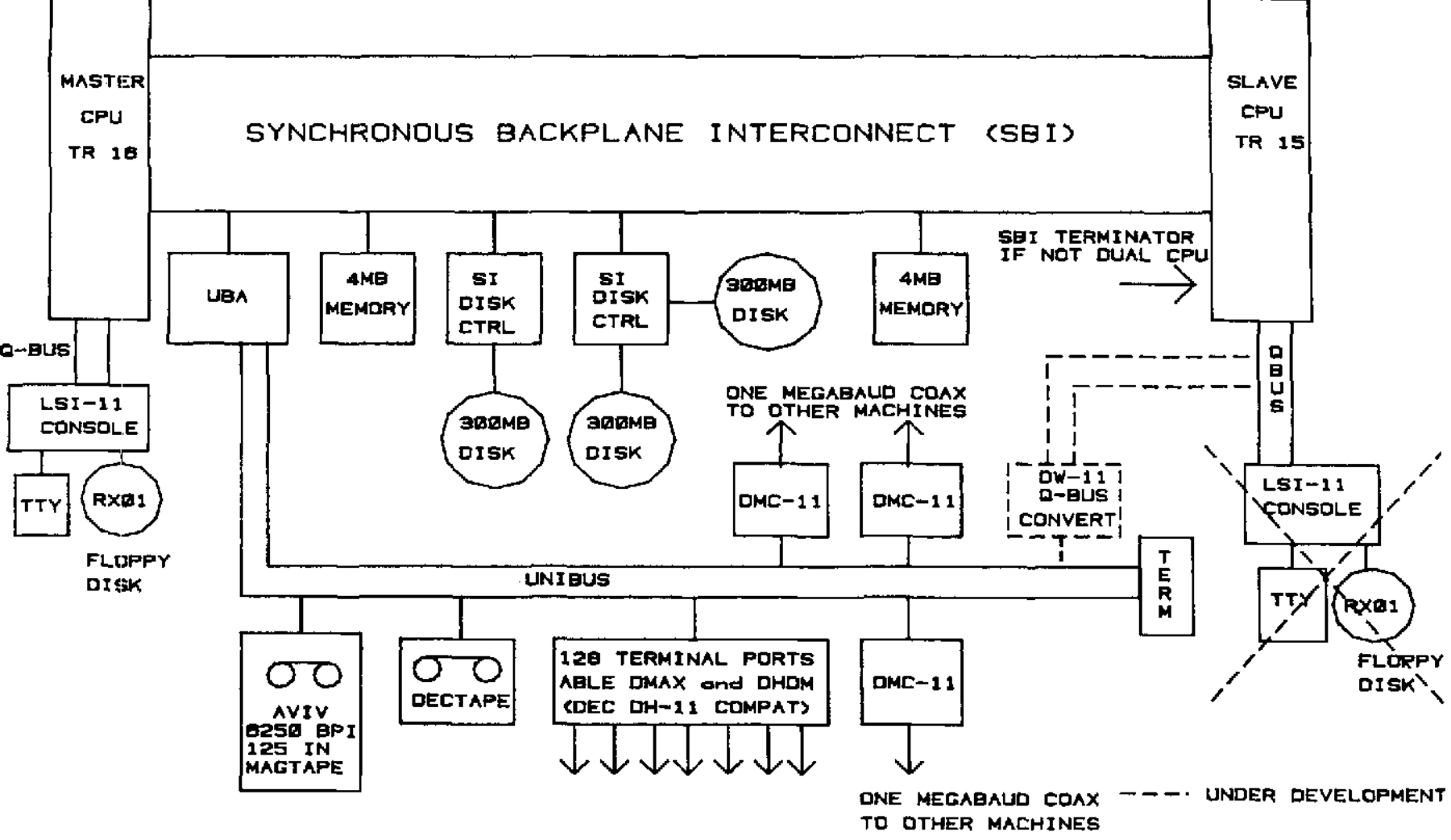
The Parchment Path?

Is there ever a time when learning is not of value—for its own sake?

Dear KV,

I've spent much of my career working as a developer and software architect for various companies, both startups and large, established firms. I've always been lucky enough to work on projects that were interesting to me, several on the cutting edge of computing technology. A lot of the interesting projects have been close to research, but I've always been on the implementation side rather than the research part. Recently, I have been inclined to poke more into the research, but I don't have the pedigree to lead such projects, as they often require an advanced degree. With only a decade or so left until retirement, I've found myself considering various Ph.D. programs, which I never would have when I was younger. When I finished my undergraduate degree, I really wanted to work in industry, and, like many people, pay off a lot of loans. Is it too late for me to think about this? On the one hand, it seems crazy to consider taking several years to work on a Ph.D. just to be able to cross the line from development to research, but on the other, maybe it's not. If I went back, I'd be, well...

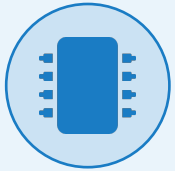
A Very Mature Student


MASTER
CPU
TR 16
SYNCHRONOUS BACKPLANE INTERCONNECT (SBI)
SLAVE
CPU
TR 15
SBI TERMINATOR
IF NOT DUAL CPU
UBA
4MB
MEMORY
SI
DISK
CTRL
SI
DISK
CTRL
300MB
DISK
4MB
MEMORY
Q-BUS
LSI-11
CONSOLE
TTY
RX01
FLOPPY
DISK
300MB
DISK
300MB
DISK
ONE MEGABAUD COAX
TO OTHER MACHINES
DMC-11
DMC-11
DW-11
Q-BUS
CONVERT
Q
B
U
S
LSI-11
CONSOLE
TTY
RX01
FLOPPY
DISK
UNIBUS
T
E
R
M
AVIV
6250 BPI
125 IN
MAGTAPE
DECTAPE
128 TERMINAL PORTS
ABLE DMAX and DHDM
(DEC DH-11 COMPAT)
DMC-11
ONE MEGABAUD COAX
TO OTHER MACHINES
UNDER DEVELOPMENT

Some (of my) research questions…

- How can system software better match modern hardware?
  - **Stop pretending every computer is a PDP-11!!!**
    - Timothy Roscoe's excellent rant upper right.
    - Kode Vicious less excellent rant lower left.
- What are the sources of performance loss due to the mismatch between software and hardware?
- How can we effectively manage code at runtime to take advantage of modern hardware?
  - Hypothesis: Hardware performance counters are an effective signal to drive program scheduling in a system with an FPGA. They are a low overhead way to find out what is going on with currently executing code.

Heterogenous Compute (All in One Box)



CPU

General Purpose

x86 / ARM

FPGA

Reconfigurable Logic

Xilinx / Intel

GPU

Parallel Processing

NVIDIA / AMD

TPU

Tensor Operations

Google / Apple

NPU

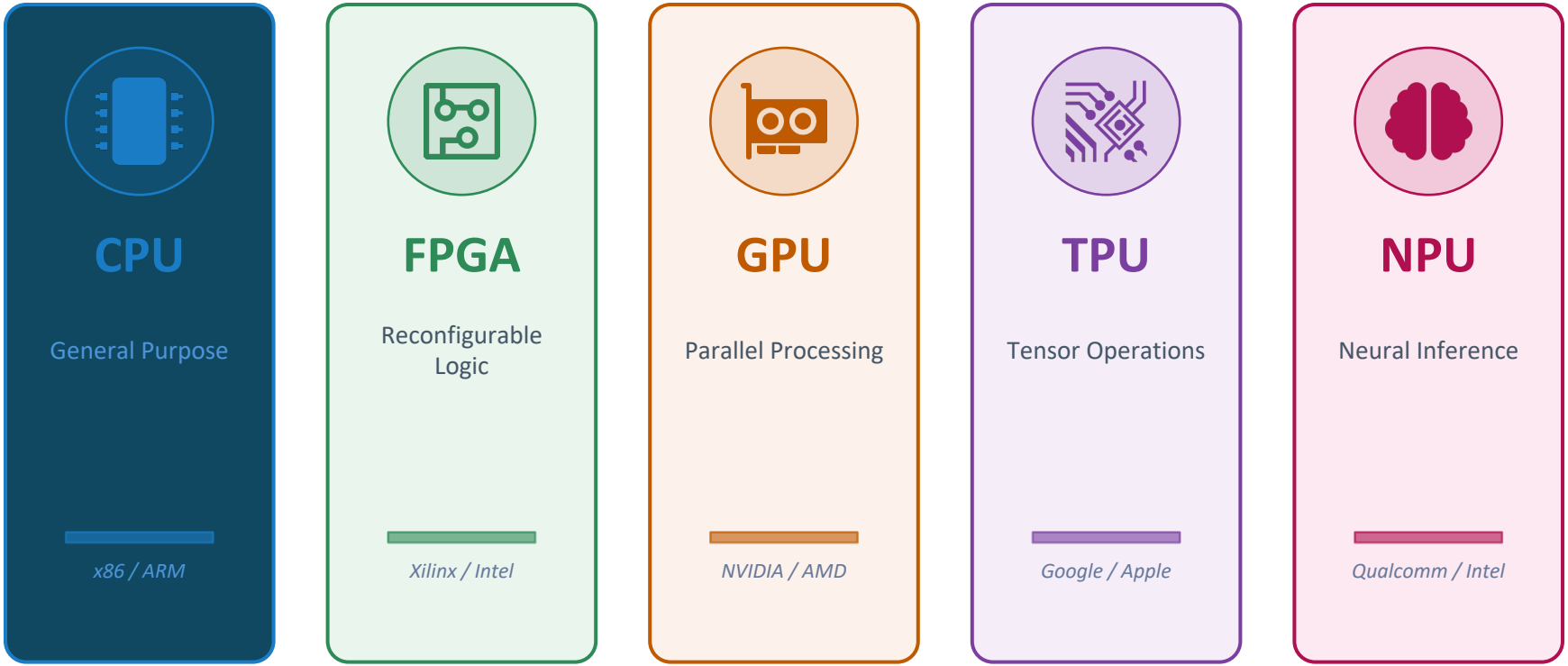
Neural Inference

Qualcomm / Intel

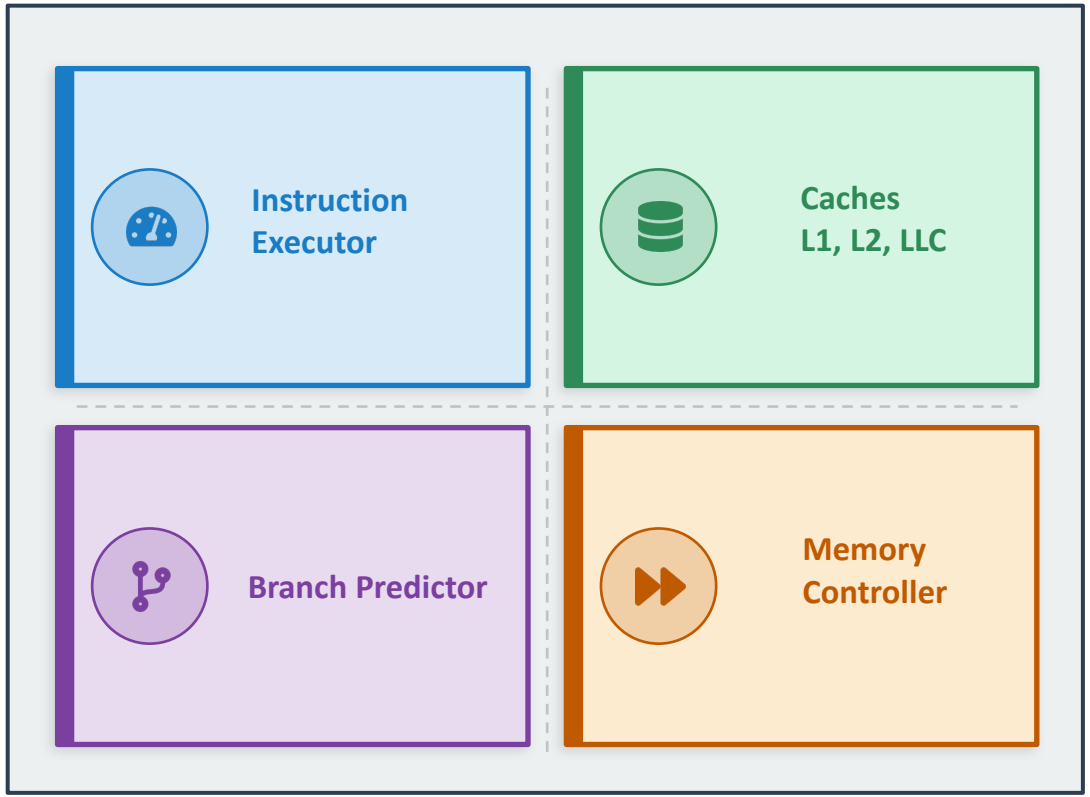
Can we move work to the optimal component?

Let’s simplify...

Let’s Look at the CPU First



Optimizing for the CPU



Why do we optimize?

- Reducing Runtime
- Reduced Latency
- Increased responsiveness
- Fair sharing of scarce resources
- Utilizing the hardware to its maximum
- Energy (more about that much later)

Intel x86-64 — Cycles Per Instruction (CPI)

Range across workload classes on modern Intel (Skylake and later)

Scenario	CPI
Theoretical max throughput (SIMD, all ports fed, no stalls)	~0.25
Dense integer/FP compute, cache-resident, no branches	0.25–1
Typical mixed workload	1–4
Memory-bound (L3 thrashing)	5–20
DRAM-bound (pointer chasing, serial misses)	50–300+
Branch-misprediction dominated	10–50

Best-case to worst-case spans ~3 orders of magnitude. SPECint geometric mean IPC (5–6 on Alder/Raptor Lake P-cores) sits between typical mixed and compute-bound — not a physical limit.

Intel x86-64 — Branch Misprediction Penalty

Sandy Bridge (2011) → Meteor Lake (2024)

Microarchitecture	Generation	Penalty (cycles)
Sandy Bridge / Ivy Bridge	2011–2013	~15
Haswell / Broadwell	2013–2015	~15–17
Skylake / Kaby / Coffee	2015–2019	~15–20
Ice Lake / Tiger Lake	2019–2021	~15–20
Alder Lake / Raptor Lake	2021–2023	~15–20
Meteor Lake / Arrow Lake	2023–2024	~15–20

Primary sources: Agner Fog's microarchitecture guide, Intel Optimization Manual, and empirical work by Nanobench/uiCA authors https://www.agner.org/optimize/instruction_tables.pdf etc.

Intel x86-64 — Cache Miss Latency

Load-to-use cycles, unloaded, per microarchitecture generation

Level	Sandy Bridge	Haswell	Skylake	Ice Lake	Alder Lake (P)
L1 hit	4	4	4	5	5
L2 hit	12	12	12	13	14
L3 hit	26–35	34–36	42–47	42–50	47–60
DRAM	150–200	180–200	200–250	200–260	200–300

ARMv8 — Branch Misprediction Penalty

Cortex, Neoverse, and Apple Silicon

Core	Context	Year	Penalty (cycles)
Cortex-A53	In-order, mobile	2013	~7–10
Cortex-A57	OOO, mobile	2013	~15
Cortex-A72	Mobile / embedded	2015	~15
Cortex-A75/A76	Mobile flagship	2017–18	~11–15
Cortex-A77/A78	Mobile flagship	2019–20	~15
Cortex-X1/X2/X3	High-perf mobile	2020–22	~15–20
Neoverse N1	Server (Graviton 2)	2019	~15
Neoverse N2/V1	Server (Graviton 3)	2021–22	~15–20
Apple M1 (Firestorm)	Desktop / laptop	2020	~15–20
Apple M2/M3	Desktop / laptop	2022–23	~15–20

In-order cores (A53, A55) benefit from shallow pipeline. Apple's wide OOO machines have excellent predictor accuracy — frequency of miss matters more than per-miss cost.

ARMv8 — Cache Miss Latency

Load-to-use cycles, unloaded

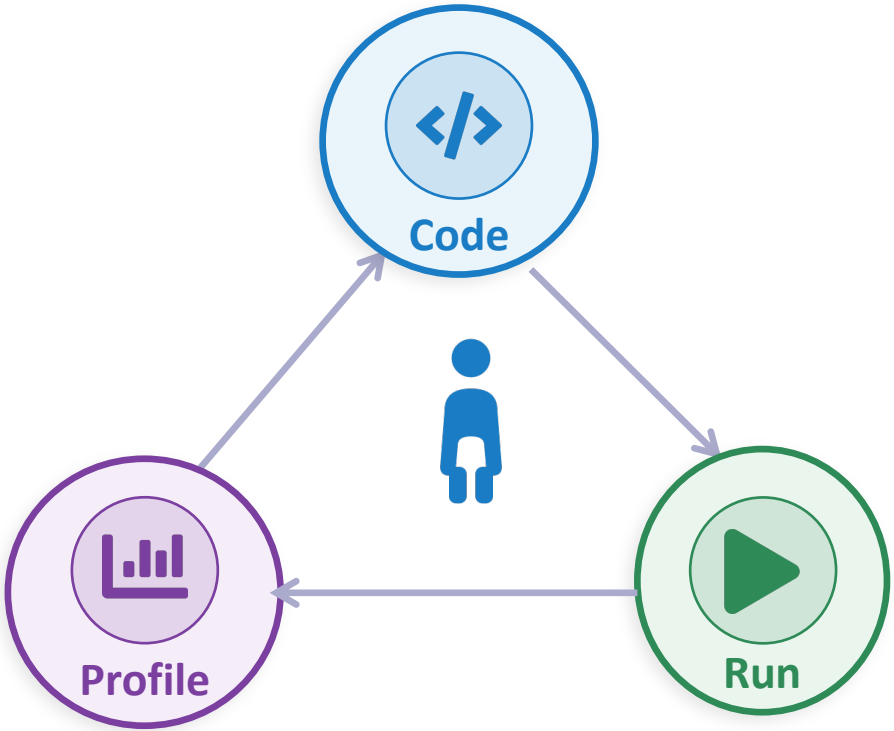
Core	L1	L2	L3 / SLC	DRAM
Cortex-A53	4	10–12	—	~150
Cortex-A57	4	16–20	—	~150–180
Cortex-A72	4	16–20	—	~150–180
Cortex-A76	4	11–13	30–40	~180–220
Cortex-A78/X1	4	11–13	30–45	~180–230
Neoverse N1	4	11–13	35–40	~180–220
Neoverse N2/V2	4	11–14	35–45	~180–230
Apple M1 (P-core)	4	~12	40–55	~80–110
Apple M2/M3	4	~12	40–55	~80–110

Apple M1/M2 DRAM appears low in cycles due to on-package LPDDR5 at high bandwidth — absolute ns latency is similar to other LPDDR5. SLC latency is SoC-dependent; DSU implementations vary by vendor.

Can we improve performance by driving the software with the hardware?

Historical Optimization Loop

A static, offline process — no runtime feedback
Always a Human in the Loop



Traditional Optimzation Tools

Intel VTune

Sampling-based profiler with deep Intel microarchitecture integration. Reports hotspots, memory access patterns, and threading bottlenecks — but tied to Intel hardware and requires a human analyst to act on results.

pmcstat + DTrace (FreeBSD)

hwpmc(4) provides low-overhead PMC counting and sampling. DTrace enables dynamic tracing at kernel and userspace boundaries. The programmer decides what to measure and what to do about it.

perf + eBPF (Linux)

perf records PMC samples and call graphs. eBPF lets programs attach arbitrary measurement logic at kernel events.

A Lot of Guesswork

All of these tools produce data. None of them act on it. A human reads the profile, forms a hypothesis, changes the code or configuration, recompiles, and re-runs..

Plus ça change…

- *Designers of compilers and instructors of computer science usually have comparatively little information about the way in which programming languages are actually used by typical programmers.*
- ***We think we know what programmers generally do but our notions are rarely based on a representative sample of the programs which are actually being run on computers.***
- *Since compiler writers must prepare a system capable of translating a language in all its generality, it is easy to fall into the trap of assuming that complicated constructions are the norm when in fact they are infrequently used.*
- ***There has been a long history of optimizing the wrong things,*** *using elaborate mechanisms to produce beautiful code in cases that hardly ever arise in practice, while doing nothing about certain frequently occurring situations.*
- *The fact that arithmetic expressions usually have an average length of only two operands, in practice, would have been a great shock to the author at that time!*
- *There has been widespread realization that more data about language use is needed; we cannot really compare two different compiler algorithms until we understand the input data.*

Source: *Knuth, D. E.* (1971). An Empirical Study of FORTRAN Programs. *Software—Practice and Experience, 1*(1), 105–133.

And what happens when the workload changes?

Different Mix of Programs

A server running a static workload is rare. Processes come and go, priorities shift, and co-located workloads compete for the same hardware resources — invalidating any static optimization.

Changing Mix of Data

Branch prediction, cache utilization, and memory access patterns all depend on the data being processed. A function optimized for one workload may be poorly tuned for the next.

New Data from the Network or Other Devices

Incoming traffic, sensor feeds, and I/O events continuously reshape the hot path. A profile taken five minutes ago will not reflect what the hardware is doing right now.

Asynchrony!

Interrupts, timers, signals, and concurrent threads mean the performance profile is always in flux.

Optimizations May Clash

- Better cache locality may disrupt branch prediction
- Improved branch prediction may harm cache locality
- Instruction scheduling may not be cache line aware
- Etc. etc. etc.

Profile Guided Optimization

Currently Best in Breed of Automated Optimization Techniques

1 Instrument

Compile with
-fprofile-instr-generate

```
clang -O2 -fprofile-instr-generate ¥
      -o prog.instr prog.c
```

2 Run

Execute on
representative input

```
./prog.instr < input.dat
llvm-profdata merge -o prog.profdata ¥
    default.profraw
```

3 Recompile

Compile with
-fprofile-instr-use

```
clang -O3 -fprofile-instr-
use=prog.profdata ¥
      -o prog.opt prog.c
```

What PGO Enables

Inlining	Hot call sites inlined based on real call frequency
Code layout	Hot basic blocks placed contiguously — fewer icache misses
Branch hints	Likely/unlikely branches annotated from observed frequencies
Loop unrolling	Unroll factor tuned to observed trip counts
Register alloc	Hot variables kept in registers across hot paths
Dead code elim	Cold paths aggressively eliminated or moved out-of-line

Runtime Reoptimization

Live Profiling

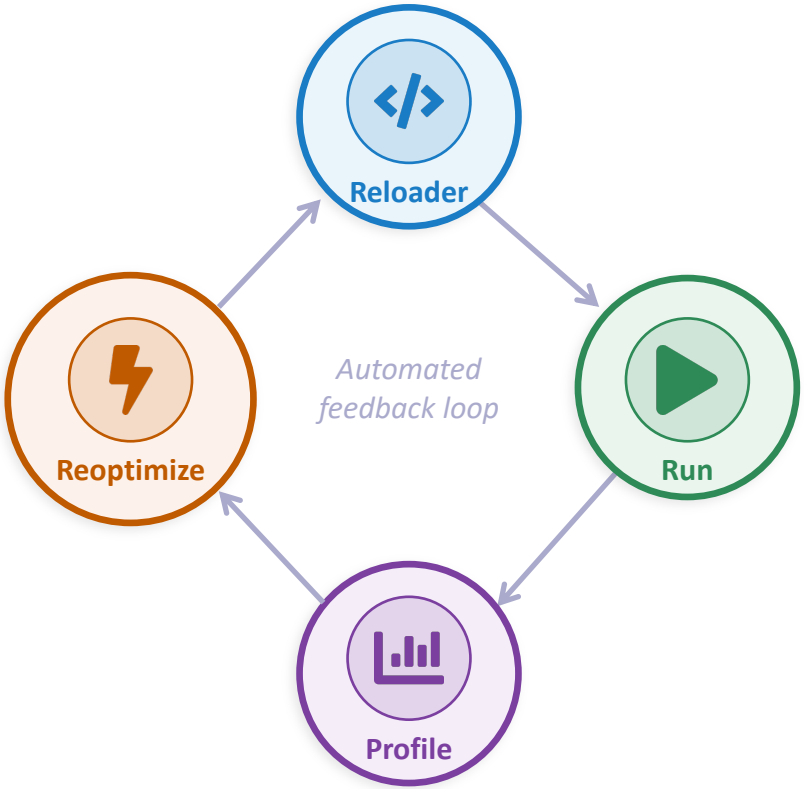
Hardware counters sample real execution — cache misses, branch mispredictions, cycles/instruction

Automatic Feedback

Profile data feeds directly into the optimizer — no human in the loop

In-place Code Reload

Hot functions are recompiled and swapped live — no restart required



Requirements for Runtime Reoptimization

1 A Stable, Independent Representation Format

The program must be stored in an architecture-neutral intermediate representation — one that can be recompiled, re-optimized, and patched at runtime without access to the original source. LLVM IR is the canonical choice: it is well-specified, mature, and retains enough semantic structure for meaningful optimization.

2 Ability to Monitor Program Performance on the CPU

The system must be able to observe — cache misses, branch mispredictions, instruction throughput, stall cycles. Hardware Performance Monitoring Counters (PMCs) provide this visibility with near-zero overhead.

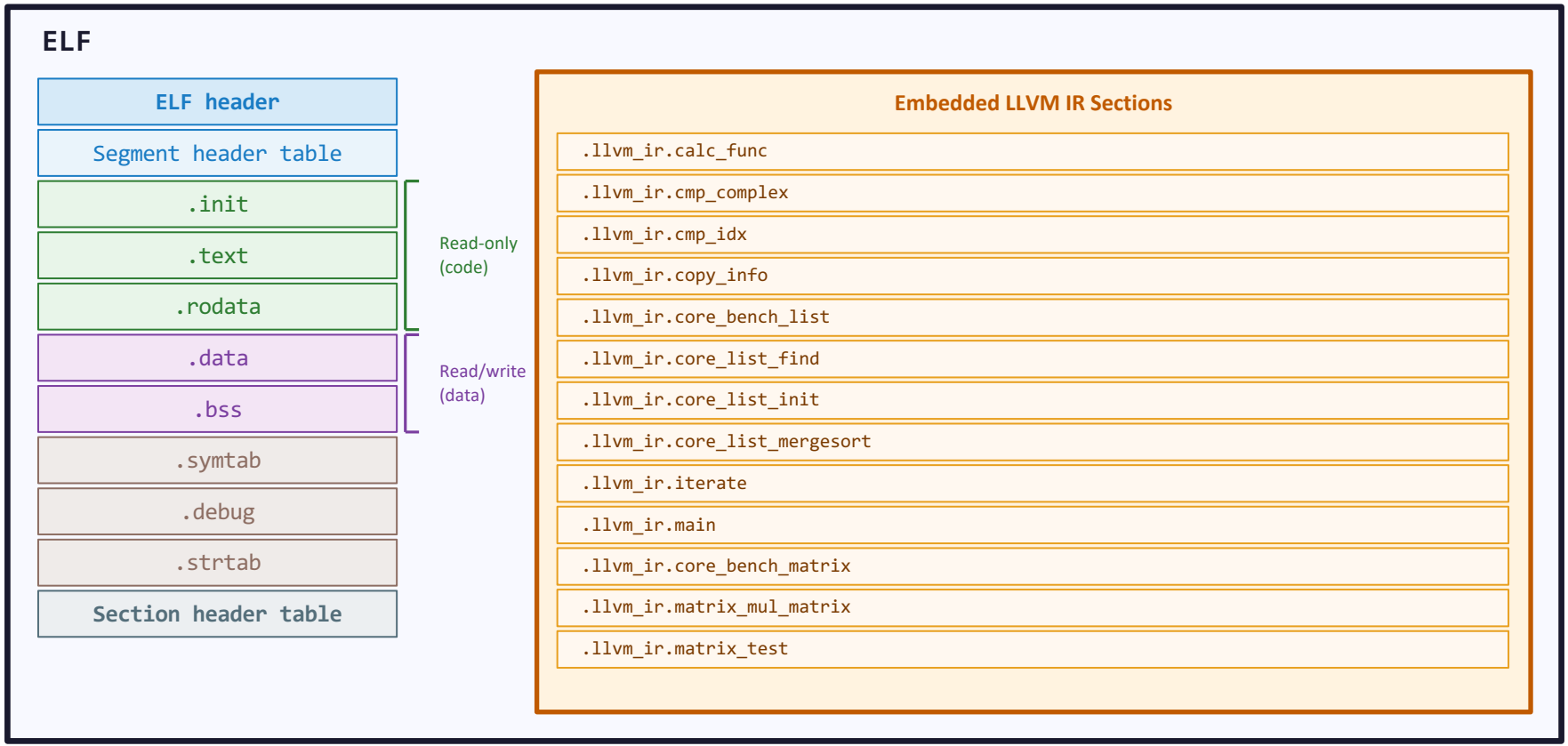
Source: *Kistler, T. & Franz, M.* (2003). Continuous Program Optimization: A Case Study. ACM Transactions on Programming Languages and Systems (TOPLAS), 25(4), 500–548.

Comparing the Approaches

	PGO	Runtime Reoptimization
Profile Source	Training run	Live counters (PMC)
When Applied	Compile time	During execution
Adapts to Input	No	Yes
Recompiles	Once, offline	On-demand, by function
Overhead	Training run only	Sampling + recompile
Correctness Risk	Low	Higher (hot patching)

Chunky Binaries

Coremark Benchmark as a Chunky Binary



Cost of Chunky Binaries

CoreMark Benchmark (~2000 LoC)

30 KB → 190 KB with Embedded IR

6× Increase in Size

30 KB → 86 KB with Compression

Size vs. Time Tradeoffs Apply

Side Benefits of Chunky Binaries

1. **Portability Across Architectures**
   IR is architecture-neutral. Can target x86-64, ARM, RISC-V, etc..

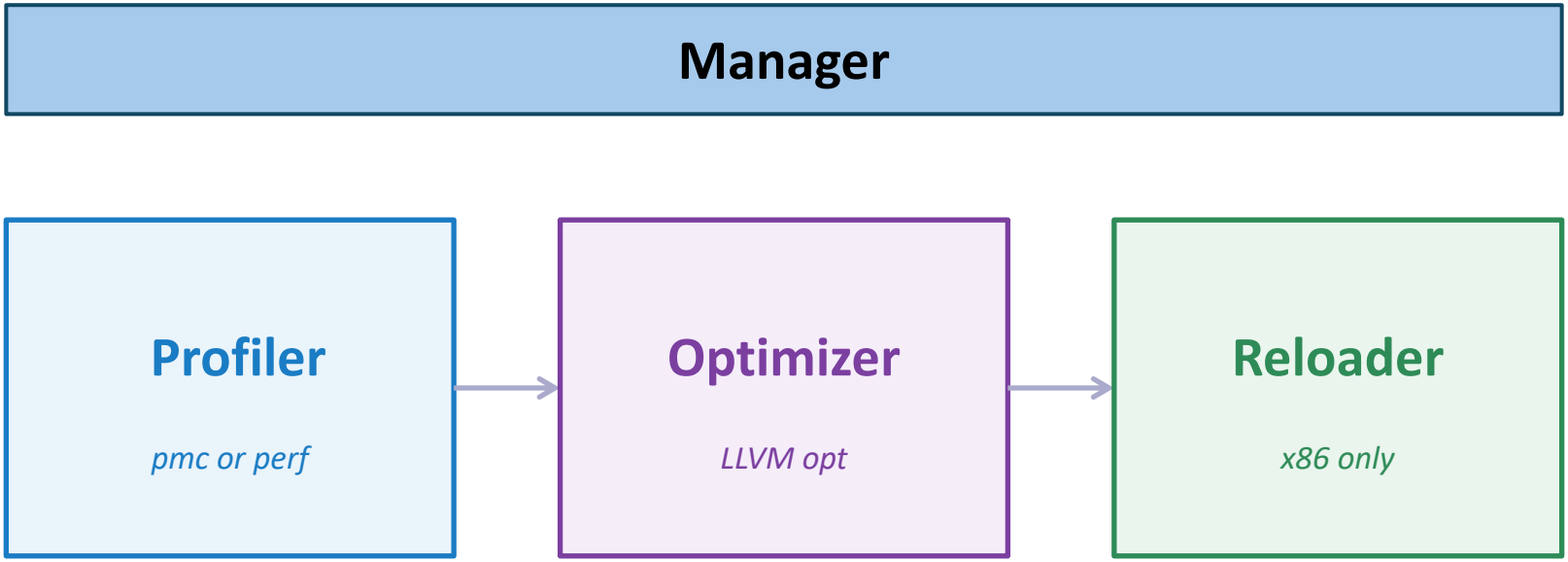
2. **New Optimizations at No Extra Cost**
   As LLVM gains new passes, existing binaries can benefit retroactively.

3. **Easily Compressed for Constrained Systems**
   LLVM IR compresses extremely well. For embedded and memory-constrained targets, the IR can be stored compressed.

4. **Signable as Part of a Supply Chain**
   The IR is a stable, inspectable artifact that can be cryptographically signed at build time.

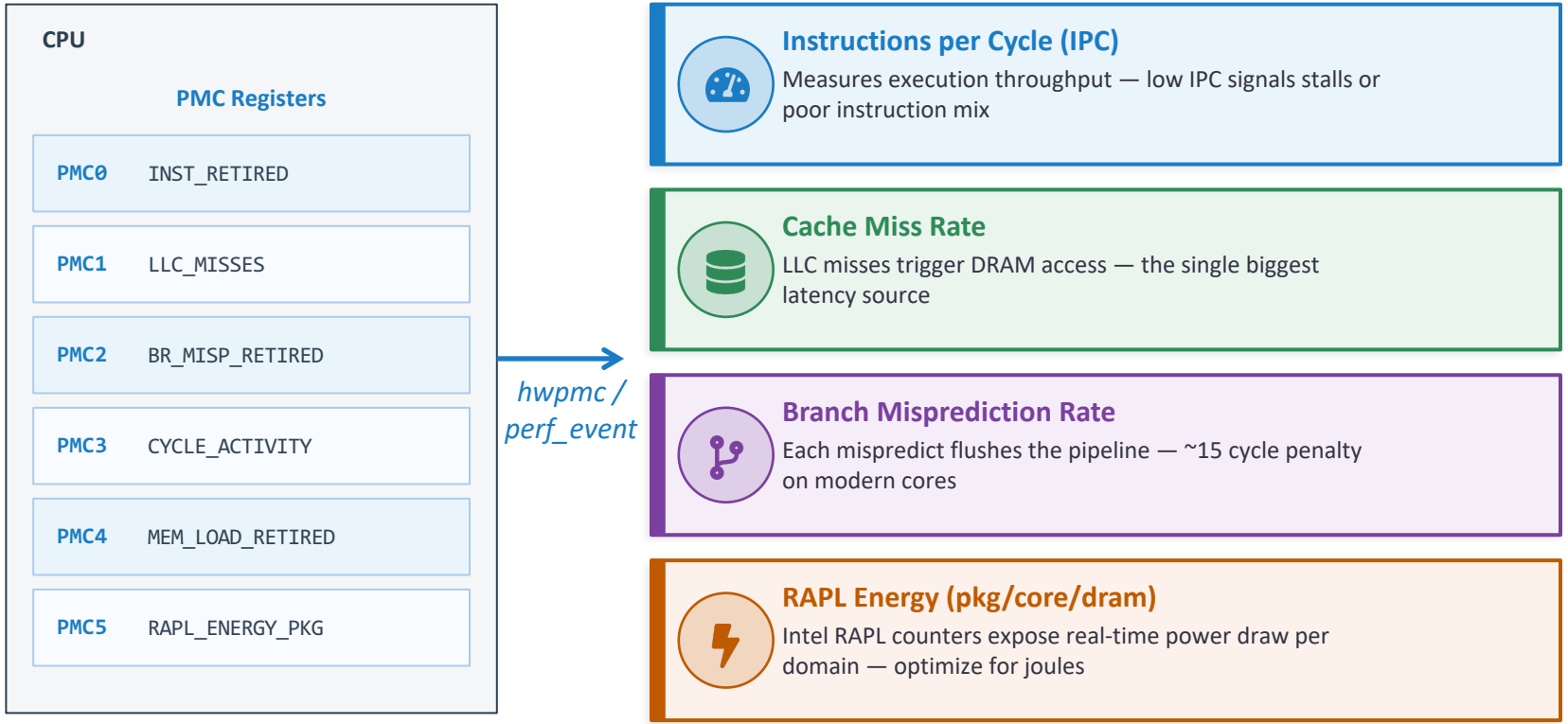
5. **Reproducible Builds**
   IR is deterministic given the same source and compiler flags.

Runtime Reoptimizer



Performance Monitoring Counters

Hardware registers that count low-level CPU events — with zero overhead



PMC Counting vs. Sampling

	COUNTING	SAMPLING
Question	How much did this happen over the whole run?	Where in the code is this happening?
Mechanism	Counter increments on every event. Read at start and end — subtract.	Counter counts down from N. On overflow, CPU interrupts and records the instruction pointer.
Output	A single integer: total event count for the interval.	A histogram of hot instruction addresses — a profile.
Overhead	Near zero. Reading a register. No interrupts, no context switches.	Proportional to sample rate. Higher frequency = more interrupts = more overhead.

LLVM Optimization Passes

Scalar

mem2reg	licm
sroa	early-cse
instcombine	reassociate
simplifycfg	tailcallelim
dce	jump-threading
dse	correlated-prop
gvn	aggr-instcombine
sccp	instsimplify

Loop

loop-rotate	loop-interchange
loop-unroll	loop-load-elim
loop-unswitch	loop-simplifycfg
loop-deletion	loop-data-pref
loop-idiom	loop-versioning
loop-reduce	indvars
loop-distribute	loop-instsimplify
loop-fusion	

Vectorization

- loop-vectorize
- slp-vectorize
- load-store-vectorizer
- vector-combine
- mve-tail-predication

Interprocedural

inliner	prune-eh
argpromotion	function-attrs
deadargelim	rpo-fn-attrs
globalopt	called-value-prop
globaldce	openmpopt
ipsccp	attributor

Memory

- memcpyopt
- dse
- mldst-motion
- gvn-hoist
- gvn-sink
- speculative-execution
- newgvn
- early-cse-memssa

Polly (Polyhedral)

- polly-canonicalize
- polly-detect
- polly-scops
- polly-opt-isl
- polly-codegen
- polly-ast
- polly-dependences

Some Early Results

- Using PMCs to drive a program’s memory usage
- Speeding up a benchmark during execution
- Searching the space of optimizations and performance counters

Detecting Cache Thrashing

A simple test of PMCs driving a program

trash.c Simulates Bad Behavior

Allocates N pages and randomly touches one byte per page in a loop — a worst-case L2 cache thrasher. Linked against libnavel for live PMC access.

PMC Threshold Triggers Shrink

If mem_load_uops_retired.l2_miss exceeds the limit in any 250ms second interval, the program calls realloc() to halve its footprint — 64 MB → 32 MB → 16 MB — until L2 misses stabilize.

Cooperative Multi-Program Sharing

Three concurrent instances compete for the same L2. Without coordination, each detects its own excess and backs off, collectively converging to stable cache usage.

```
$ ./trash 64 mem_load_uops_retired.l2_miss 200000

Allocating 64 MB (16384 pages)
handler total: 303210   interval: 303210
COUNT HIGH! → reallocating to 32 MB
handler total: 306658   interval: 306658
COUNT HIGH! → reallocating to 16 MB
handler total: 2168     interval: 2168
handler total: 174719   interval: 172551
handler total: 361557   interval: 186838
handler total: 535976   interval: 174419  ✓
```

Known Limits & What's Next

- Threshold requires manual tuning today
- No upward reallocation once stable
- No code reoptimization only memory

Why Functions?

1

Functions Are Easy to Hot Patch

A function has a single well-known entry point. Replacing it requires patching only the first few bytes with a jump to new code.

2

Natural Boundary

Functions are the natural unit of compilation, linking, and ABI. Reoptimizing at function granularity keeps the rest of the program unchanged and avoids invalidating unrelated code.

3

Sampling Techniques Report on Functions

PMC sampling via pmcstat and perf attribute counter events to functions

4

Large Enough to Be a Good Optimization Target

Functions contain the patterns that optimization passes are designed to improve.

CoreMark

An industry-standard CPU benchmark focused on real-world embedded workloads

Linked List

Insert, find, and sort operations — tests pointer chasing and branch prediction

`core_list_init` `core_list_find`

Contributes ~25% of total CoreMark score

Matrix Math

3×3 integer matrix multiply and add — tests arithmetic throughput and cache reuse

`core_bench_matrix`

Contributes ~25% of total CoreMark score

Score = iterations/sec

Higher is better.
Must run ≥ 10 seconds.
Results must be reproducible.

State Machine

Regular expression state transitions on a data stream — tests branch-heavy control flow

`core_bench_state`

Contributes ~25% of total CoreMark score

CRC Computation

16/32-bit CRC over data buffers — tests integer ops, shifts, and XOR patterns

`crc16` `crcu32` `crcu8`

Contributes ~25% of total CoreMark score

Key Facts

Created by	EEMBC (2009)
Language	ANSI C
Iterations	~2000 / run
Data size	Fits in L1/L2
Parallelism	Single / multi
Deterministic	CRC validation

Why CoreMark?

- Hot functions are well-defined
- IR sections map 1:1 to functions
- Portable, no OS dependencies

Inlining threshold reshapes CoreMark's micro-architecture profile

% change from inline-threshold-10 baseline across 5 PMC categories

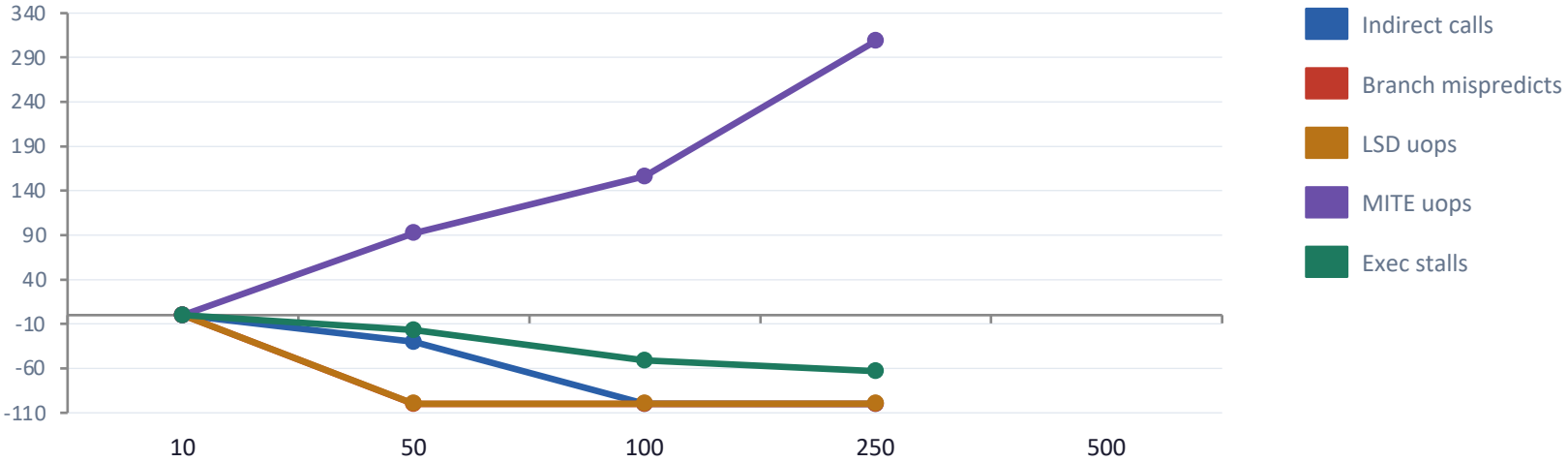
Indirect calls	
10	baseline
50	+0%
100	-30%
250	-100%
500	-100%

Branch mispredicts	
10	baseline
50	+0%
100	-100%
250	-100%
500	-100%

LSD uops	
10	baseline
50	+0%
100	-100%
250	-100%
500	-100%

MITE uops	
10	baseline
50	+0%
100	+92%
250	+156%
500	+309%

Exec stalls	
10	baseline
50	+0%
100	-17%
250	-51%
500	-63%



X-axis = inline threshold; Y-axis = % change vs inline10 baseline. Negative = reduction, positive = increase.

PMC category summary: inline250 is the overall best optimization

Counter category	inline50	inline100	inline250	inline500	
Indirect branches	-22.2%	-96.1%	**-98.8%**	-97.9%	*inline250 near-eliminates*
Direct calls	-11.2%	-25.3%	-83.3%	**-87.0%**	*continues improving to inline500*
Branch mispredicts	-10.3%	**-72.5%**	-67.4%	-26.8%	*inline50 already wins*
LSD (loop buffer)	-99.6%	**-99.8%**	-99.8%	-99.8%	*dead at inline50*
MITE uop fetch	**+92.4%**	+156.0%	+308.6%	+164.0%	*cost: grows with inlining*
Execution stalls	-17.0%	-51.0%	**-63.1%**	-63.1%	*inline250 delivers most gain*
FP divide ops	-0.9%	-3.1%	**-74.4%**	-74.4%	*abrupt drop at inline250*
DTLB walk misses	-53.9%	-53.1%	**-84.1%**	-84.1%	*inline250 clears most walks*

= best value in row

= best value where increase = bad

Polly — Polyhedral Loop Optimizer

SCoP Requirements

- Affine loop bounds
- Affine array subscripts
- No irregular control flow

Key Optimizations

Loop tiling	Blocking for cache
Loop interchange	Data locality
Loop fusion	Merge loops
Loop fission	Split for parallelism
Auto-parallelism	Emit OpenMP
Auto-vectorization	Exploit SIMD
Data locality	Reduce cache misses

Analysis Passes

Pass	Purpose	IC	DCL
`polly-detect`	Detects valid SCoPs	—	—
`polly-scops`	Builds polyhedral repr.	—	—
`polly-deps`	Computes data deps	—	—
`polly-ast`	Generates AST from schedule	—	—
`polly-fn-scops`	Lists SCoPs in function	—	—

Transformation Passes

Pass	Purpose	IC	DCL
`polly-simplify`	Simplifies SCoP repr.	Lo	—
`polly-optree`	Forwards operand trees	Me	Lo
`polly-delicm`	Promotes mem → scalars	Hi	Hi
`polly-prune-unprof`	Skips unworthy SCoPs	Lo	—
`polly-opt-isl`	**Core: optimal schedule (ISL)**	**Hi**	**Hi**
`polly-codegen`	Emits optimized LLVM IR	Hi	Hi

Usage with opt

Quick:

```
opt -O3 -polly ¥
    input.ll -o out.ll
```

Full pipeline:

```
opt -polly-canonicalize ¥
    input.ll -o canon.ll


opt ¥
  -polly-simplify ¥
  -polly-optree ¥
  -polly-delicm ¥
  -polly-simplify ¥
  -polly-prune-unprofitable ¥
  -polly-opt-isl ¥
  -polly-codegen ¥
  canon.ll -o opt.ll
```

PolyBench/C

The polyhedral benchmark suite — 30 kernels designed to stress loop optimizers like Polly

Linear Algebra

Matrix operations with perfectly nested loops — ideal SCoP candidates

`gemm` `gemver` `gesummv`

Stencils

Iterative grid computations — expose data locality and tiling opportunities

`jacobi-1d` `jacobi-2d`

Datamining

Covariance, correlation, and PCA-style reductions over large arrays

`covariance` `correlation`

Solvers & Transforms

LU decomposition, Cholesky, Durbin, Gramschmidt — triangular loop nests

`cholesky` `durbin`

Metric: FLOPS or wall-clock time

Time single kernel or full suite.
Compare -O0, -O3, -O3 -polly.

Key Facts

Kernels	30 total
Language	ANSI C
Dataset	MINI → EXTRALARGE
Loop depth	2–6 levels
Arrays	Stack or heap
Timing	POLYBENCH_TIME

Why PolyBench?

- Known SCoP structure
- Validates IR roundtrip
- Isolates loop opt effects

Pessimization via the Optimizer

jacobi-2d: applying Polly loop optimization doubles wall-clock runtime

```
=== jacobi-2d_base ===

4.60 real
4.59 user
0.01 sys
```

```
=== jacobi-2d_polly ===

8.00 real
7.99 user
0.00 sys
```

Raw Results

jacobi-2d: Base vs. Polly — selected PMC counters

PMC Counter	Base	Polly	Δ%	Interpretation
inst_retired.any_p	10,170,760,904	10,681,502,487	+5%	Only 5% more instructions — Polly does similar work
uops_executed.cycles_ge_2_uops_exec	2,329,827,853	5,237,777,280	+124%	Executor running at higher utilization — good
idq.all_dsb_cycles_4_uops	230,169,829	1,466,490,545	+537%	Decoded stream buffer at full width — good
uops_executed.core_cycles_ge_1	7,206,875,681	10,066,905,374	+40%	More active cycles — executing more per unit time
mem_uops_retired.all_loads	3,806,039,134	5,476,566,656	+44%	More loads — expected with wider loop bodies
offcore_requests.outstanding.demand_code	9,003,950,730	45,304,687,385	+403%	L3 demand code fetches ↑ — footprint may have grown
br_inst_exec.all_branches	430,289,163	2,470,511,846	+474%	Far more branches executed — loop structure changed
br_misp_exec.taken_conditional	9,422	1,053,040	+11,076 %	Branch mispredictions ↑↑ — 15–20 cycle penalty each
br_misp_retired.near_taken	9,435	1,365,771	+14,376 %	Confirms misprediction spike — needs investigation
br_inst_retired.nontaken_conditional	5,769,322	986,784,985	+17,004 %	Conditional branch explosion — optimizer added branches

Positive — executor working harder / better utilization

Notable — worth monitoring; not necessarily harmful

Concern — branch mispredictions increased dramatically

Polly Pessimization

Baseline vs. Polly-optimized — key counter deltas

✓ Executor Running Hot

+5.0%	`inst_retired.any_p`	Barely more instructions — low overhead
+37.1%	`idq.all_dsb_cycles_4_uops`	Decoder delivering max 4 uops/cycle more often
+124.8%	`uops_exec.cycles_ge_2_uops`	Execution units running at high occupancy
0→11326	`move_elim.simd_eliminated`	Polly adding SIMD vector eliminations

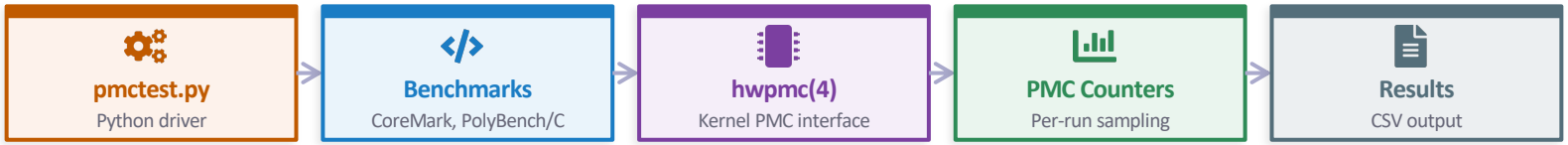
Pessimization

+5403%	`offcore_rqsts.code_rd`	L3 instruction fetches exploded — code footprint bloat
+74.2%	`br_inst_exec.all_branches`	Far more branches from loop transforms
+76.4%	`br_misp_exec.taken_cond`	New branch patterns mispredicting (15–20 cycle hit each)
+105.9%	`resource_stalls.rs`	Reservation station pressure doubled

Net: Polly makes the executor run harder but inflates code size.

Benchmarking the Benchmarks

pmctest.py — runs each benchmark under every PMC counter and collects results



What pmctest.py does

1. **Enumerate PMC events**
   Queries the kernel for all available hardware counter names on the current CPU
2. **For each benchmark**
   Runs each benchmark, with command line options
3. **Run benchmark under pmcstat or perf**
   Uses PMCs in counter mode to count events
4. **Reports counter**
   Pmcstat or perf reports count after the process exits
5. **Repeat for every counter**
   Re-runs the benchmark once per PMC event — each run is independent
6. **Emit results**
   Writes counter values, one per counter name in /tmp/counter_name files

Counter Name	inline10
lsd.cycles_4_uops	1556323
lsd.cycles_active	990986
br_misp_exec.taken_indirect_near_call	1418152
br_inst_exec.taken_indirect_near_call	132589186

Type of Events Sampled

INST_RETIRED	Total instructions
CPU_CLK_UNHALTED	Active cycles
LLC_MISSES	Last-level cache misses
BR_MISP_RETIRED	Branch mispredictions
MEM_LOAD_RETIRED	Memory loads retired

Some Open Questions

1

Optimization Overhead

Recompiling and reloading a function takes time. How large is that cost relative to the gains? At what speedup threshold does reoptimization become net-positive?

2

Switching Frequency

How often should a hot function be re-evaluated for reoptimization? Too frequent wastes compile budget; too infrequent misses behavioral shifts.

3

Sampling Frequency

PMC sampling itself has overhead. How fine-grained must sampling be to detect meaningful counter changes? Can we adapt sampling rate ?

Mom Always Said...

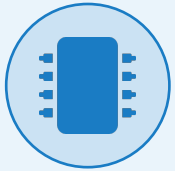
“Never do a live demo!”

Next Steps

- Narrow benchmark selection
- Choose 3 real-world workloads
- Table-driven optimizations
- ML optimizer
- Extend rr-mgr to include more runtime statistics
  - ~~Time to reoptimize~~
  - Running reoptimization score

Other Execution Targets

Runtime reoptimization can target any compute architecture



CPU

General Purpose

x86 / ARM

FPGA

Reconfigurable Logic

Xilinx / Intel

GPU

Parallel Processing

NVIDIA / AMD

TPU

Tensor Operations

Google / Apple

NPU

Neural Inference

Qualcomm / Intel

Intel x86-64 — Energy Cost per Microarchitectural Event

Marginal dynamic energy, Haswell–Skylake era. Derived via RAPL regression across workloads.

Event	Energy (joules)	Notes
Retired instruction (simple ALU)	0.3–1 nJ	Marginal dynamic cost; fixed core power dominates
L1 hit	~0 (marginal)	Already paid by instruction energy
L1 miss → L2	1–5 nJ	
L2 miss → L3	5–20 nJ	
L3 miss → DRAM	50–200 nJ	~60 pJ/bit at DRAM + controller; 64B line ≈ 60–100 nJ at DRAM alone
Branch misprediction	10–40 nJ	~15–20 wasted cycles × marginal energy/cycle

Khan et al. (2018) — "RAPL in Action": https://dl.acm.org/doi/10.1145/3177754 (ACM TOMPECS, Vol. 3 No. 2)

Performance = Power

Power = Performance

If you needed something to work on...

- Machine Learning Libraries
- Update hwpmc for RAPL
- Power Aware Scheduler
- Performance Aware Scheduler

Questions?

Comments?