CesASMe and Staticdeps: static detection of memory-carried dependencies for code analyzers

Théophile Bastian, Hugo Pompougnac, Alban Dutilleul, Fabrice Rastello

CesASMe and Staticdeps: static detection of memory-carried dependencies for code analyzers

Théophile Bastian
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
38000 Grenoble, France
theophile.bastian@inria.fr

Hugo Pompougnac
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
38000 Grenoble, France
hugo.pompougnac@inria.fr

Alban Dutilleul
ENS Rennes
France
alban.dutilleul@ens-rennes.fr

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
38000 Grenoble, France
fabrice.rastello@inria.fr

Abstract

A variety of code analyzers, such as `IACA`, `uiCA`, `llvm-mca` or `Ithemal`, strive to statically predict the throughput of a computation kernel. Each analyzer is based on its own simplified CPU model reasoning at the scale of a basic block. Facing this diversity, evaluating their strengths and weaknesses is important to guide both their usage and their enhancement.

We present `CesASMe`, a fully-tooled solution to evaluate code analyzers on C-level benchmarks composed of a benchmark derivation procedure that feeds an evaluation harness. We conclude that memory-carried data dependencies are a major source of imprecision for these tools. We tackle this issue with `staticdeps`, a static analyzer extracting memory-carried data dependencies, including across loop iterations, from an assembly basic block. We integrate its output to `uiCA`, a state-of-the-art code analyzer, to evaluate `staticdeps`' impact on a code analyzer's precision through `CesASMe`.

1 Introduction

At a time when software is expected to perform more computations, faster and in more constrained environments, tools that statically predict the resources they consume are very useful to guide optimizations. This need is reflected in the diversity of binary or assembly code analyzers that appeared following the deprecation of `IACA` [11], which Intel has maintained through 2019. Whether it is `llvm-mca` [24], `uiCA` [3], `Ithemal` [16] or `Gus` [10], all these tools strive to produce various performance metrics, including the number of CPU cycles a computation kernel will take — which roughly translates to execution time. In addition to raw measurements relying on hardware counters, these model-based analyses provide higher-level and refined data, to expose the bottlenecks and guide the optimization of a given code. This feedback is useful to experts optimizing computation kernels, including scientific simulations and deep-learning kernels.

An exact throughput prediction would require a cycle-accurate simulator of the processor such as gem5 [6], but would also require microarchitectural data that is most often not publicly available, and would be prohibitively slow in any case. These tools thus each solve in their own way the challenge of modeling complex CPUs while remaining simple enough to yield a prediction in a reasonable time, ending up with different models.

In particular, previous works (*eg.* `AnICA` [22]) have exposed significant discrepancies in the reported predictions for programs with memory-carried dependencies. This is a clue of a more general limitation of current code analyzers, which leads to the hypothesis that the underlying performance models have trouble with memory dependencies modelization. One way to verify this is to provide an alternative analysis method and be able to show that it produces more accurate predictions. Such an approach will also need an evaluation framework sound enough to trustworthy compare one method with another.

1.1 Contributions

We present a new method to construct sound baselines against which to compare the predictions of code analyzers, in order to evaluate the performance models underlying the latter. Indeed, existing approaches lack such a baseline and often require *ad hoc*, handmade supplementary analysis. Some evaluate the results of a code analyzer by comparing it to those of another code analyzer (*eg.* `AnICA`), while others perform actual measurements, but on — possibly randomly-generated — basic blocks isolated from their context, *eg.* using `BHive` [8].

By comparison, our approach, which is materialized in a tool called `CesASMe`, consists in generating various (L1-resident, code analyzers-compliant) *microbenchmarks* from an existing benchmark suite, and then using them without isolating their constituent basic blocks to evaluate code analyzers. To do so, `CesASMe` applies static analyses on their constituent basic blocks. At the same time, it executes the microbenchmarks and performs hardware counters-based measurements. It finally aggregates measurements and analyses in such a way as to make them commensurable.

We also provide a new heuristic called `staticdeps` to extract memory-carried dependencies. We use this heuristic with `CesASMe` to test the hypothesis mentioned above that

current code analyzers struggle to model the latter. We show that this approach significantly improves the accuracy of the code analyzer uiCA[3], which is tantamount to showing that dependencies constitute a significant limitation of its underlying model. We also show that the resulting extended analyzer fully outperforms the state of the art.

1.2 Related works

The static throughput analyzers studied rely on a variety of models. IACA [11], developed by Intel and now deprecated, is closed-source and relies on Intel's expertise on their own processors. The LLVM compiling ecosystem, to guide optimization passes, maintains models of many architectures. These models are used in the LLVM Machine Code Analyzer, `llvm-mca` [24], to statically evaluate the performance of a segment of assembly. Independently, Abel and Reineke used an automated microbenchmark generation approach to generate port mappings of many architectures in `uops.info` [1, 2] to model processors' backends. This work was continued with uiCA [3], extending this model with an extensive frontend description. Following a completely different approach, Ithemal [16] uses a deep neural network to predict basic blocks throughput. To obtain enough data to train its model, the authors also developed BHive [8], a profiling tool working on isolated basic blocks.

Another static tool, Osaca [14], provides lower- and upper-bounds to the execution time of a basic block. As this kind of information cannot be fairly compared with tools yielding an exact throughput prediction, we exclude it from our scope.

All these tools statically predict the number of cycles taken by a piece of assembly or binary that is assumed to be the body of an infinite — or sufficiently large — loop in steady state, all its data being L1-resident. As discussed by uiCA's authors [3], hypotheses can subtly vary between analyzers; *eg.* by assuming that the loop is either unrolled or has control instructions, with non-negligible impact. Some of the tools, *eg.* Ithemal, necessarily work on a single basic block, while some others, *eg.* IACA, work on a section of code delimited by markers. However, even in the second case, the code is assumed to be *straight-line code*: branch instructions, if any, are assumed not taken.

Throughput prediction tools, however, are not all static. Gus [10] dynamically predicts the throughput of a whole program region, instrumenting it to retrieve the exact events occurring through its execution. This way, Gus can detect bottlenecks more finely through sensitivity analysis, at the cost of a significantly longer run time.

The BHive profiler [8] takes another approach by performing basic block throughput *measurement* instead of analysis: by mapping memory at any address accessed by a basic block, it can effectively run and measure arbitrary code without context, often — but not always, as we discuss later — yielding good results as one would expect from an actual measurement.

The AnICA framework [22] also attempts to evaluate throughput predictors by finding examples on which they are inaccurate. AnICA starts with randomly generated assembly snippets, and refines them through a process derived from abstract interpretation to reach general categories of problems.

2 Benchmarking harness

The obvious solution to evaluate predictions is to compare them to an actual measure. However, as these tools reason at the basic block level, this is not as trivially defined as it would seem, especially when it comes to showing the impact of dependencies. As detailed later in Section 2.6, *without proper context, a basic block's throughput is not well-defined.*

To recover the context of each basic block, we reason instead at the scale of a C source code. This makes the measures unambiguous: one can use hardware counters to measure the elapsed cycles during a loop nest. This requires a suite of benchmarks, in C, that both is representative of the domain studied, and wide enough to have a good coverage of the domain. However, this is not in itself sufficient to evaluate static tools: the number of cycles reported by the counters can be the result of an arbitrarily large number of loop turns, with conditional branches taken or not. This number hardly compares to `llvm-mca`, IACA, Ithemal, and uiCA basic block-level predictions seen above that only predict throughput of individual basic block.

Predictions lifting. We developed a *basic block occurences counter* to count the number of occurrences of a basic block during kernel execution. Then, we use this number of occurrences to weight basic block throughput predictions. Finally, we lift code analyzers' results by summing the weighted basic block predictions:

$$\text{lifted\_pred}(\mathcal{K}) = \sum_{b \in \text{BBs}(\mathcal{K})} \text{occurences}(b) \times \text{pred}(b) \quad (1)$$

Architecture of our benchmarking harness. Our benchmarking framework works in four successive stages, whose a high-level view is shown in Figure 1. It first generates relevant *microbenchmarks* (*ie.* L1-resident benchmarks) by applying a series of transformations to an existing benchmark suite which is, in our case, the HPC-native Polybench [20] suite. It then extracts the basic blocks constituting a computation kernel, and instruments it to retrieve their respective occurrences in the original context. It runs all the studied tools on each basic block, while also running measures on the whole computation kernel. Finally, the block-level results are lifted to kernel-level results thanks to the occurrences previously measured.

2.1 Generating microbenchmarks

The first stage of our benchmarking harness consists in generating a set of *microbenchmarks*. We extend AnICA's

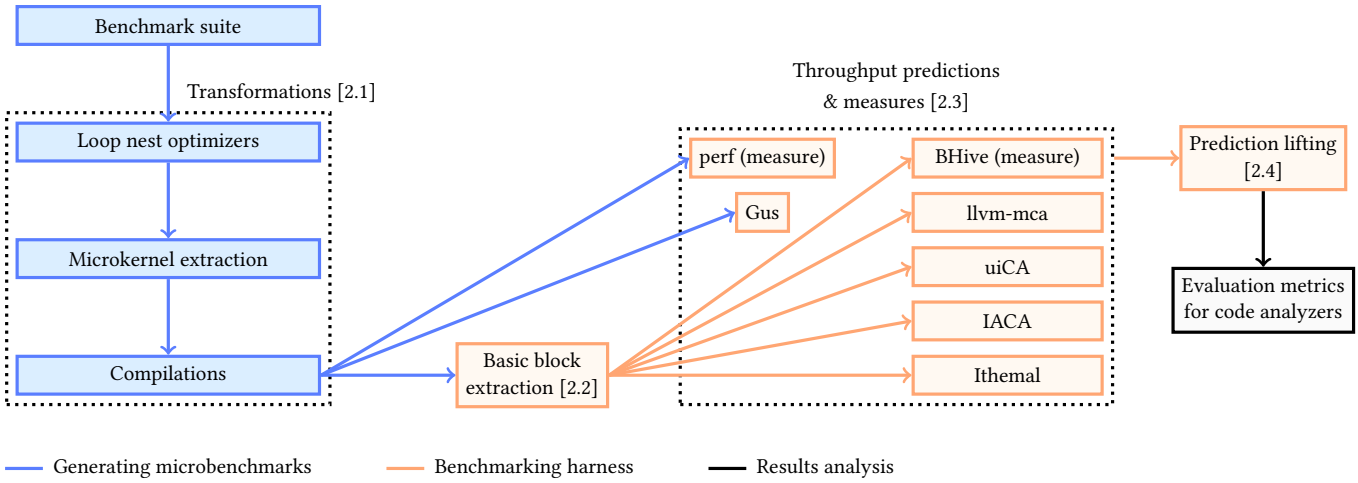

Figure 1. Our analysis and measurement environment.

random-based approach mentioned above in order to produce microbenchmarks diverse enough, but related to real-life kernels. Thus, our approach consists in applying several *transformations* to an existing benchmark suite:

1. We produce several versions of each benchmark using loop nest optimizers (Pluto[7] and PoCC [19]). The latter gives access to a wide variety of transformations: register tiling, tiling, skewing, vectorization/simdization, loop unrolling, loop permutation, loop fusion and so on.
2. We instrument the resulting code in order to extract the loop which has the greatest impact on performance while being L1-resident. This *microkernel* is isolated from the initial benchmark, and incorporated into a parameterized template where requested behaviors, as the number of successive measurements, are set. We call this refined benchmark a *microbenchmark*.
3. Given a microbenchmark, we produce several binaries by applying different compilation strategies (enabling/disabling auto-vectorization, extended instruction sets, *etc.*).

This pipeline allows us to produce a diversified yet domain-specific range of microbenchmarks.

2.2 Basic block extraction

Given a compiled microbenchmark, we use the Capstone disassembler [21], and split the assembly code at each control flow instruction (jump, call, return, ...) and each jump site.

To accurately obtain the occurrences of each basic block in the whole kernel's computation, we then instrument it with `gdb` by placing a break point at each basic block's first instruction in order to count the occurrences of each basic block between two calls to the `perf` counters[1].

2.3 Throughput predictions and measures

The harness leverages a variety of tools: actual CPU measurement; the `BHive` basic block profiler [8]; `llvm-mca` [24], `uiCA` [3] and `IACA` [11], which leverage microarchitectural models to predict a block's throughput; `Ithemal` [16], a machine learning model; and `Gus` [10], a dynamic analyzer based on `QEmu` that works at the whole binary level.

The execution time of the full kernel is measured using Linux `perf` [15] CPU counters around the full computation kernel. The measure is repeated four times and the smallest is kept; as the caches are not flushed, this ensures that the cache is warm and compensates for context switching or other measurement artifacts. `Gus` instruments the whole function body. The other tools work at basic block level; these are run on each basic block of each benchmark.

We emphasize the importance, throughout the whole evaluation chain, to keep the exact same assembled binary. Indeed, recompiling the kernel from source *cannot* be assumed to produce the same assembly kernel. This is even more important in the presence of slight changes: for instance, inserting `IACA` markers at the C-level — as is intended — around the kernel *might* change the compiled kernel, if only for alignment reasons. Furthermore, those markers prevent a binary from being run by overwriting registers with arbitrary values. This forces a user to run and measure a version which is different from the analyzed one. In our harness, we circumvent this issue by adding markers directly at the assembly (already compiled) basic block level. Our `gdb` instrumentation procedure also respects this principle of single-compilation. As `QEmu` breaks the `perf` interface, we have to run `Gus` with a preloaded stub shared library to be able to instrument binaries containing calls to `perf`.

2.4 Prediction lifting and filtering

We finally lift single basic block predictions to a whole-kernel cycle prediction by summing the block-level results, weighted by the occurrences of the basic block in the original context (Equation 1 above). If an analyzer fails on one of the basic blocks of a benchmark, the whole benchmark is discarded for this analyzer.

[1]We assume the program under analysis to be deterministic.

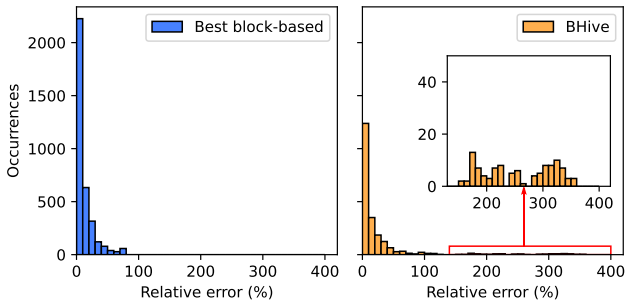

Figure 2. Relative error distribution *wrt.* `perf`

Finally, we control the proportion of cache misses in the program's execution using `Cachegrind` [17] and `Gus`; programs that have more than 15 % of cache misses on a warm cache are not considered L1-resident and are discarded.

2.5 Soundness of `CesASMe`'s methodology

Given a time prediction C_{pred} (in cycles) and a baseline C_{baseline}, we define its relative error as

$$\text{err} = \frac{\left|C_{\text{pred}} - C_{\text{baseline}}\right|}{C_{\text{baseline}}} \tag{2}$$

We assess the commensurability of the whole benchmark, measured with `perf`, to lifted block-based results by measuring the statistical distribution of the relative error of two series: the predictions made by `BHive`, and the series of the best block-based prediction for each benchmark.

We single out `BHive` as it is the only tool able to *measure* — instead of predicting — an isolated basic block's timing. Since `BHive` is based on measures — instead of predictions — through hardware counters, an excellent accuracy is expected. This, however, is not sufficient: as discussed later in Section 2.6, `BHive` is not able to yield a result for about 40 % of the benchmarks, and is subject to large errors in some cases.

The result of this analysis is presented in Table 1 and in Figure 2. The results are in a range compatible with common results of the field, as seen *eg.* in [3] reporting Mean Absolute Percentage Error (MAPE, corresponding to the "Average" row) of about 10-15 % in many cases. While lifted `BHive`'s average error is driven high by large errors on certain benchmarks, investigated later in this article, its median error is still comparable to the errors of state-of-the-art tools. From this, we conclude that lifted cycle measures and predictions are consistent with whole-benchmark measures; and consequently, lifted predictions can reasonably be compared to one another.

2.6 Understanding `BHive`'s results

The error distribution of `BHive` against `perf`, plotted right in Figure 2, puts forward irregularities in `BHive`'s results. Its lack of support for control flow instructions can be held accountable for a portion of this accuracy drop; our lifting method, based on block occurrences instead of paths, can explain another portion. We also find that `BHive` fails to produce a result in about 40 % of the kernels explored — which means that, for those cases, `BHive` failed to produce a result on at least one of the constituent basic blocks. In fact, this is due to the need to reconstruct the context of each basic block *ex nihilo*.

	Best block-based	**BHive**
Datapoints	3500	2198
Errors	0	1302
	(0 %)	(37.20 %)
Average (%)	11.60	27.95
Median (%)	5.81	7.78
Q1 (%)	1.99	3.01
Q3 (%)	15.41	23.01

Table 1. Relative error statistics *wrt.* `perf`

The basis of `BHive`'s method is to run the code to be measured, unrolled a number of times depending on the code size, with all memory pages but the code unmapped. As the code tries to access memory, it will raise segfaults, caught by `BHive`'s harness, which allocates a single shared-memory page, filled with a repeated constant, that it will map wherever segfaults occur before restarting the program. `BHive` mainly fails on bad code behaviour (*eg.* control flow not reaching the exit point of the measure if a bad jump is inserted), too many segfaults to be handled, or a segfault that occurs even after mapping a page at the problematic address.

The registers are also initialized, at the beginning of the measurement, to the fixed constant `0x2324000`. We show through two examples that this initial value can be of crucial importance.

The following experiments are executed on an Intel(R) Xeon(R) Gold 6230R CPU (Cascade Lake), with hyperthreading disabled.

Imprecise analysis. Consider this x86-64 kernel:

```
vmulsd (%rax), %xmm3, %xmm0
vmovsd %xmm0, (%r10)
```

When executed with all the general purpose registers initialized to the default constant, `BHive` reports 9 cycles per iteration, since `%rax` and `%r10` hold the same value, inducing a read-after-write dependency between the two instructions. If, however, `BHive` is tweaked to initialize `%r10` to a value that aliases (*wrt.* physical addresses) with the value in `%rax`, *eg.* between `0x10000` and `0x10007` (inclusive), it reports 19 cycles per iteration instead; while a value between `0x10008` and `0x1009f` (inclusive) yields the expected 1 cycle — except for values in `0x10039-0x1003f` and `0x10079-0x1007f`, yielding 2 cycles as the store crosses a cache line boundary.

In the same way, the value used to initialize the shared memory page can influence the results whenever it gets loaded into registers.

Failed analysis. Some memory accesses will always result in an error; for instance, it is impossible (by default) to `mmap` at an address lower than `0x10000`. Thus, with equal initial values for all registers, the following kernel would fail, since the second operation attempts to load at address 0:

```
subq %r11, %r10
movq (%r10), %rax
```

Such errors can occur in more circumvoluted ways. The following x86-64 kernel, for instance, is extracted from a version of the `durbin` kernel:

```
vmovsd 0x10(%r8, %rcx), %xmm6
subl %eax, %esi
movslq %esi, %rsi
vfmadd231sd -8(%r9, %rsi, 8), \
    %xmm6, %xmm0
```

Here, `BHive` fails to measure the kernel when run with registers initialized to the default constant at the 2nd occurrence of the unrolled loop body, failing to recover from an error at the `vfmadd231sd` instruction with the `mmap` strategy. Indeed, after the first iteration the value in `%rsi` becomes null, then negative at the second iteration; thus, the second occurrence of the last instruction fetches at address `0xffffffff0a03ff8`, which is in kernel space. This microkernel can be benchmarked with BHive *eg.* by initializing `%rax` to 1. Furthermore, on most 64-bits CPUs, addresses are only valid if they are representable on 48 bits (correctly sign-extended to fill 64 bits) [4, 12], called *canonical form*, causing further failures in `BHive`. Other kernels still fail with accesses relative to the instruction pointer, as `BHive` read-protects the unrolled microkernel's instructions page.

3 Static extraction of memory-carried dependencies

As put forwards by `AnICA` [22], memory-carried dependencies are a difficult challenge for code analyzers. This is further confirmed by the results of `CesASMe`, presented in Section 4.3.

To this end, we present `staticdeps`, a heuristic-based static assembly analyzer able to extract most of the memory-carried dependencies that are relevant from a latency analysis perspective. It operates at the basic-block level, and tracks dependencies across arbitrarily complex pointer arithmetic, through arbitrarily many loop iterations.

Working at the basic-block level, it is however lacking context, as we argued earlier; as such, it cannot detect aliasing stemming from outside of the loop.

3.1 Relevance of memory dependencies in hardware

Modern Out-of-Order (OoO) hardware has dedicated significant architectural resources to maximize the pipeline usage. One such key component is the reorder buffer (ROB) — an extension of Tomasulo's algorithm [25] —, a circular buffer storing incoming μOPs, decoded and renamed from instructions, in the frontend. When a μOP is committed, as soon as it is no longer busy, the ROB can safely move its head pointer, freeing space for new μOPs at its tail.

In particular, if a μOP μ_1 is not yet committed, the ROB may not contain μOPs more than the ROB's size ahead of μ_1. This is also true for instructions, as the vast majority of instructions decode to at least one μOP[2].

Nevertheless, instruction level parallelism (ILP) might still be limited, either by dependencies or other factors. Speculative techniques such as memory disambiguation address that point, by for instance trying to detect whether a given load depends on an earlier not-yet-committed store, in order to allow safe out-of-order execution of non-dependent memory accesses. If such speculation turns out wrong, execution is typically rolled back to before the offending load in the ROB, incurring at least the penalty of another load.

Recent microarchitectures still lack ways of dealing with true data dependencies, especially at the memory level, even if ongoing research with value prediction [18, 23] tackles that issue, but to the best of our knowledge value prediction has not yet been implemented in products of mainstream CPU vendors.

3.2 Detecting different types of dependencies

Depending on their type, some dependencies are significantly harder to statically detect than others.

Register-carried dependencies in straight-line code are easiest to detect, by keeping track of which instruction last wrote each register in a *shadow register file*. This is most often supported by code analyzers — for instance, `llvm-mca` and `uiCA` support it.

Register-carried, loop-carried dependencies can, to some extent, be detected the same way. As the basic block is always assumed to be the body of an infinite loop, a straight-line analysis can be performed on a duplicated kernel. This strategy is *eg.* adopted by `Osaca` [14] (§II.D). When dealing only with register accesses, this strategy is always sufficient: as each iteration always executes the same basic block, it is not possible for an instruction to depend on another instruction two iterations earlier or more.

Memory-carried dependencies, however, are significantly harder to tackle. While basic heuristics can handle some simple cases, in the general case two main difficulties arise:

(i) pointers may *alias*, *ie.* point to the same address or array; for instance, if `%rax` points to an array, it may be that `%rbx` points to `%rax` + 8, making the detection of such a dependency difficult;
(ii) arbitrary arithmetic operations may be performed on pointers, possibly through diverting paths: *eg.* it might be necessary to detect that `%rax` + 16 << 2 is identical to `%rax` + 128/2; this requires semantics for assembly

[2]Some `mov` instructions from register to register may, for instance, only have an impact on the renamer; no μOPs are dispatched to the backend.

instructions and tracking formal expressions across register values — and possibly even memory.

Tracking memory-carried dependencies is, to the best of our knowledge, not done in code analyzers, or only with simple heuristics, as our results with `CesASMe` suggests.

For loop-carried, memory-carried dependencies, the strategy previously used for register-carried dependencies is not sufficient at all times when the dependencies tracked are memory-carried, as dependencies may reach further back than the previous iteration, *eg.* in a dynamic implementation of the Fibonacci sequence.

3.3 The `staticdeps` heuristic

To statically detect memory dependencies, we use a randomized heuristic approach. We consider the set $\mathcal{R}$ of values representable by a 64-bits unsigned integer; we extend this set to $\overline{\mathcal{R}} = \mathcal{R} \cup \{\bot\}$, where $\bot$ denotes an invalid value. We then proceed by keeping track of which instruction last wrote each memory address written to, using the following principles.

- Whenever an unknown value is read, either from a register or from memory, generate a fresh value from $\mathcal{R}$, uniformly sampled at random. This value is saved to a shadow register file or memory, and will be used again the next time this same data is accessed. Note that a shadow is a data structure that describes the original component with a given granularity and additional meta-information. For instance, a shadow memory in our case is a set of memory locations, each of which is associated with a value from $\overline{\mathcal{R}}$ and the last instruction that wrote to it.
- Whenever an integer arithmetic operation is encountered, compute the result of the operation and save the result to the shadow register file or memory.
- Whenever another kind of operation or an unsupported operation is encountered, save the destination operand as $\bot$; this operation is assumed to not be valid pointer arithmetic. Operations on $\bot$ always yield $\bot$ as a result.
- Whenever writing to a memory location, compute the written address using the above principles, and keep track of the instruction that last wrote to a memory address.
- Whenever reading from a memory location, compute the read address using the above principles, and generate a dependency from the current instruction to the instruction that last wrote to this address (if known).

To handle loop-carried dependencies, we remark that while far-reaching dependencies may *exist*, they are not necessarily *relevant* from a performance analysis point of view. Indeed, if an instruction i_2 depends on a result previously produced by an instruction i_1, this dependency is only relevant if it is possible that i_1 is not yet completed when i_2 is considered for issuing — else, the result is already produced, and i_2 needs not wait to execute. Given the structure of the ROB as detailed in Section 3.1 above, it seems that a dependency is relevant only if it does not reach more than the size of the ROB away; we formalize and prove this intuition in Section 3.5 below.

A possible solution to detect loop-carried dependencies in a kernel $\mathcal{K}$ is thus to unroll it until it contains at least $|\text{ROB}| + |\mathcal{K}|$ instructions. This ensures that every instruction in the last kernel can find dependencies reaching up to |ROB| back.

On Intel CPUs, the reorder buffer size contained 224 μOPs on Skylake (2015), or 512 μOPs on Golden Cove (2021) [26]. These sizes are small enough to reasonably use this solution without excessive slowdown.

3.4 Practical implementation

We implement `staticdeps` in Python, using `pyelftools` and the `capstone` disassembler to extract and disassemble the targeted basic block. The semantics needed to compute encountered operations are obtained by lifting the kernel's assembly to `valgrind`'s `VEX` intermediary representation.

The implementation of the heuristic detailed above provides us with a list of dependencies across iterations of the considered basic block. We then "re-roll" the unrolled kernel: each dependency is written as a triplet $(\texttt{source}, \texttt{dest}, \Delta k)$, where the first two elements are the source and destination instruction of the dependency *in the original, non-unrolled kernel*, and Δk is the number of iterations of the kernel between the source and destination of the dependency.

Finally, we filter out spurious dependencies: each dependency found should occur for each kernel iteration i at which $i + \Delta k$ is within bounds. If the dependency is found for less than 80 % of those iterations, the dependency is declared spurious and is dropped.

3.5 Long distance dependences can be ignored

Definition 1 (Distance between instructions)**.** *Let $(I_p)_{0 \le p < n}$ be the trace of executed instructions. For $p < q$, distance $(I_p, I_{p'})$ is the overall number of decoded μOPs for the subtrace $(I_r)_{p \le r \le p'}$ minus one.*

Theorem 1 (Long distance dependencies)**.** *A dependency between two instructions that are separated by at least R others μOPs can be ignored.*

To prove this assertion we need a few postulates that describe the functioning of a CPU and in particular how μOPs transit in (decoded) and out (retired) the reorder buffer.

Postulate 1 (Reorder buffer as a circular buffer)**.** *Reorder buffer is a circular buffer of size say R. It contains only decoded μOPs. Let us denote i_d the μOP at position d in the reorder buffer. Assume i_d just got decoded. We have that for every q and q' in $[0, R)$:*

$$(q - d - 1)\%R < (q' - d - 1)\%R \iff i_q \text{ is decoded before } i_{q'}$$

If a μOP has not been retired yet (issued and executed), it cannot be replaced by any freshly decoded instruction. In other words every non-retired decoded μOP are in the reorder buffer. This is possible thanks to the notion of *full reorder buffer*:

Postulate 2 (Full reorder buffer)**.** *Let us denote by i_d the μOP that just got decoded. The reorder buffer is said to be full if for $q = (d+1)\%R$, μOP i_q is not retired yet. If the reorder buffer is full, then instruction decoding is stalled.*

Let $(I_p)_{0\leq p<n}$ be a trace of executed instructions. Each of these instructions are iteratively decoded, issued, and retired. We will also denote by $(i_q)_{0\leq q<m}$ the trace of decoded μOPs. To prove Theorem 1 we need to state that two in-flight μOPs are distant of at most R μOPs. For any instruction I_p, we denote as Q_p the range of indices such that $(i_q)_{q\in Q_p}$ are the μOPs from the decoding of I_p. We will call an *in-flight* μOP, a μOP that is decoded but not yet retired; It is necessarily in the reorder buffer.

Lemma 1 (Distance of in-flight μOPs)**.** *For any pair of instructions $(I_p, I_{p'})$, and two corresponding μOPs, $(i_q, i_{q'})$ with $(q, q') \in Q_p \times Q_{p'}$,*

$$inflight(i_q) \wedge inflight(i_{q'}) \Rightarrow distance\left(I_p, I_{p'}\right) < R$$

In case of branch misprediction, some additional instructions might also be decoded (and potentially issued). In other words, $(i_q)_{0\leq q<m}$ potentially contains more μOPs than those matching the executed instruction. However, if not surjective, the relation from $\{I_p\}_{0\leq p<n}$ to $\{i_q\}_{0\leq q<m}$ is clearly injective and increasing (using the order within which instructions and μOPs are listed). In other words, distance $\left(I_p, I_{p'}\right) \leq |q' - q|$. Observe that at any time, the content of the ROB can be seen as a window of length R over $(i_q)_{0\leq q<m}$. Consequently, if both i_q and $i_{q'}$ are in-flight then $|q' - q| < R$. □

Postulate 3 (Issue delay)**.** *Reasons why the issue of a μOP i is delayed can be:*

1. *i is not yet in the reorder buffer*
2. *i depends on μOP i' which is not retired yet*
3. *ports on which i can be mapped are all occupied*

Theorem 1 is now a direct consequence (proof by contradiction) of the previous observations. Let us consider a delayed issue for μOP i where the unique cause is a dependence from μOP i', that is:

1. i is already in the reorder buffer
2. i depends on μOP i' which is not retired yet
3. at least one port on which i can be mapped is available

Since i' is not retired yet and i' is "before" i, i' is still in the reorder buffer, *ie.* both i and i' are in the reorder buffer. □

3.6 Limitations

In its current implementation, `staticdeps` is limited to read-after-write dependencies, which we believe to be sufficient for latency analysis on usual CPUs. However, our heuristic may easily support other types of dependencies (*eg.* write-after-write) by tracking additional data if necessary.

We argued earlier that one of the shortcomings that most crippled state-of-the-art tools was that analyses were conducted out-of-context, considering only the basic block at hand. This analysis is also true for `staticdeps`, as it is still focused on a single basic block in isolation; in particular, any aliasing that stems from outside the analyzed basic block is not visible to `staticdeps`.

Work towards a broader analysis range, *eg.* at the scale of a function, or at least initializing values with gathered assertions — maybe based on abstract interpretation techniques — could be beneficial to the quality of dependencies detections.

As `staticdeps`'s heuristic is based on randomness, it may yield false positives: two registers could theoretically be assigned the same value sampled at random, making them aliasing addresses. This is, however, very improbable, as values are sampled from a set of cardinality 2^{64}. If necessary, the error can be reduced by amplification: running multiple times the algorithm on different randomness seeds reduces the error exponentially.

Conversely, `staticdeps` should not present false negatives due to randomness. Dependencies may go undetected, *eg.* because of out-of-scope aliasing or unsupported operations. However, no dependency that falls into the scope of our baseline `depsim` — as briefly described in Section 4.4 — should be missed because of random initialisations.

In silicon, dependencies are implemented not between instructions, but between μOPs. Modeling the decomposition of instructions to μOPs, together with providing semantics for each μOP, however, is not done to our knowledge. Furthermore, it would need to be done for each microarchitecture: while instructions' semantics are defined by the ISA, each microarchitecture may split instructions to μOPs in its own way.

4 Experiments and evaluation

Our experiments are twofold. We first analyze the results from `CesASMe`, from which we confirm `AnICA`'s case study: memory-carried dependencies are a crucial shortcoming of state-of-the-art analyzers. We then evaluate `staticdeps`' capacity to detect these dependencies: we first compare the dependencies detected to those actually present, then evaluate the gain in precision when enriching `uiCA`'s model with `staticdeps`.

4.1 Experimental environment

The experiments presented in this paper were all realized on a Dell PowerEdge C6420 machine from the Grid5000 cluster [5], equipped with 192 GB of DDR4 SDRAM — only a small fraction of which was used — and two Intel Xeon Gold 6130 CPUs (x86-64, Skylake microarchitecture) with 16 cores each.

The experiments themselves were run inside a Docker environment very close to our artifact, based on Debian Bullseye. Care was taken to disable hyperthreading to improve measurements stability. For tools whose output is based on a direct measurement (`perf`, `BHive`), the benchmarks were run sequentially on a single core with no experiments on the other cores. No such care was taken for `Gus` as, although based on a dynamic run, its prediction is purely function of recorded program events and not of program measures. All other tools were run in parallel.

We use `llvm-mca` `v13.0.1`, `IACA` `v3.0-28-g1ba2cbb`, `Gus` at commit `87463c9`, `BHive` at commit `5f1d500`, `uiCA` at commit `9cbbe93`, `Ithemal` at commit `b3c39a8`.

4.2 CesASMe: analyzers' throughput predictions

Running `CesASMe`'s harness provides us with 3500 benchmarks — after filtering out non-L1-resident benchmarks —, on which each throughput predictor is run. We make the full output of our tool available in our artifact.

The raw complete output from `CesASMe` — roughly speaking, a large table with, for each benchmark, a cycle measurement, cycle count for each throughput analyzer, the resulting relative error, and a synthesis of the bottlenecks reported by each tool — enables many analyses that, we believe, could be useful both to throughput analysis tool developers and users. Tool designers can draw insights on their tool's best strengths and weaknesses, and work towards improving them with a clearer vision. Users can gain a better understanding of which tool is more suited for each situation.

The error distribution of the relative errors, for each tool, is presented as a box plot in Figure 3. Statistical indicators are also given in Table 2. We also give, for each tool, its Kendall's tau indicator [13]: this indicator, used to evaluate *eg.* uiCA [3] and Palmed [9], measures how well the pairwise ordering of benchmarks is preserved, −1 being a full anti-correlation and 1 a full correlation. This is especially useful when one is not interested in a program's absolute throughput, but rather in comparing which program has a better throughput.

These results are, overall, significantly worse than what each tool's article presents. We attribute this difference mostly to the specificities of Polybench: composed of computation kernels, it intrinsically stresses the CPU more than basic blocks extracted out of the Spec benchmark suite. This difference is clearly reflected in the experimental section of the Palmed article [9]: the accuracy of most tools is worse on Polybench than on Spec, often by more than a factor of two.

As `BHive` and `Ithemal` do not support control flow instructions (*eg.* `jump` instructions), those had to be removed from the blocks before analysis. While none of these tools, apart from `Gus` — which is dynamic —, is able to account for branching costs, these two analyzers were also unable to account for the front- and backend cost of the control flow

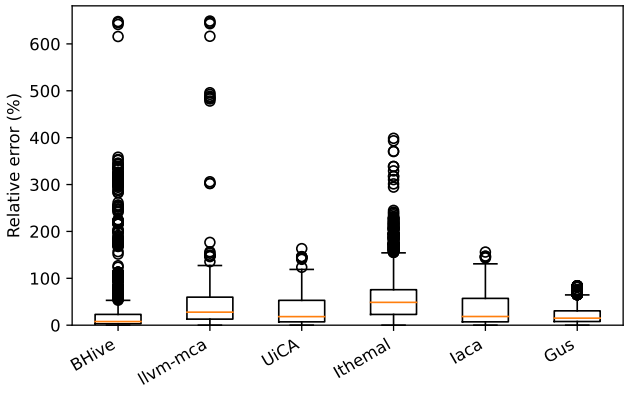

Figure 3. Statistical distribution of relative errors

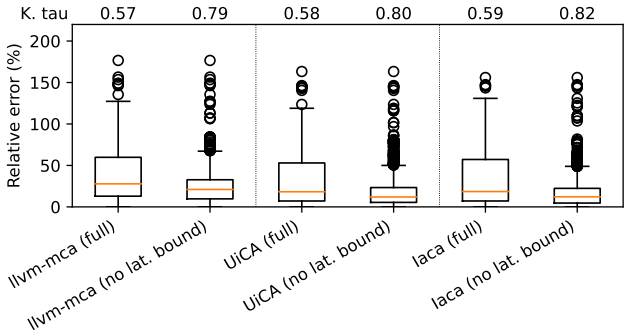

Figure 4. Statistical distribution of relative errors, with and without pruning latency bound through memory-carried dependencies rows (`llvm-mca` outliers trimmed)

instructions themselves as well — corresponding to the TP_U mode introduced by `uiCA` [3], while others measure TP_L.

4.3 CesASMe: impact of dependency-boundness

An overview of the full results table (available in our artifact) suggests that on a significant number of rows, the static tools — thus leaving `Gus` and `BHive` apart —, excepted `Ithemal`, often yield comparatively bad throughput predictions *together*. To confirm this observation, we look at the 30 % worst benchmarks — in terms of MAPE relative to `perf` — for `llvm-mca`, `uiCA` and `IACA` — yielding 1050 rows each. All of these share 869 rows (82.8 %), which we call *jointly bad rows*. In the overwhelming majority (97.5 %) of those jointly bad rows, the tools predicted fewer cycles than measured, meaning that a bottleneck is either missed or underestimated.

There is no easy way, however, to know for certain which of the 3500 benchmarks are latency bound: no hardware counter reports this. We investigate this further using `Gus`'s sensitivity analysis: in complement of the "normal" throughput estimation of `Gus`, we run it a second time, disabling the accounting for latency through memory dependencies. By construction, this second measurement should be either very close to the first one, or significantly below. We then assume

Bencher	Datapoints	Failures	(%)	MAPE	Median	Q1	Q3	K. tau	Time (CPU·h)
BHive	2198	1302	(37.20 %)	27.95 %	7.78 %	3.01 %	23.01 %	0.81	1.37
llvm-mca	3500	0	(0.00 %)	36.71 %	27.80 %	12.92 %	59.80 %	0.57	0.96
UiCA	3500	0	(0.00 %)	29.59 %	18.26 %	7.11 %	52.99 %	0.58	2.12
Ithemal	3500	0	(0.00 %)	57.04 %	48.70 %	22.92 %	75.69 %	0.39	0.38
Iaca	3500	0	(0.00 %)	30.23 %	18.51 %	7.13 %	57.18 %	0.59	1.31
Gus	3500	0	(0.00 %)	20.37 %	15.01 %	7.82 %	30.59 %	0.82	188.04

Table 2. Statistical analysis of overall results

a benchmark to be latency bound due to memory-carried dependencies when it is at least 40 % faster when this latency is disabled; there are 1112 (31.8 %) such benchmarks.

Of the 869 jointly bad rows, 745 (85.7 %) are declared latency bound through memory-carried dependencies by `Gus`. In Section 4.2, we presented in Figure 3 general statistics on the tools on the full set of benchmarks. We now remove the 1112 benchmarks flagged as latency bound through memory-carried dependencies by `Gus` from the dataset, and present in Figure 4 a comparative box plot for the tools under scrutiny, with Kendall's tau coefficient. While the results for `llvm-mca`, `uiCA` and `IACA` globally improved significantly, the most noticeable improvements are the reduced spread of the results and the Kendall's τ correlation coefficient's increase.

From this, we argue that detecting memory-carried dependencies is a weak point in current state-of-the-art static analyzers, and that their results could be significantly more accurate if improvements are made in this direction. We thus confirm `AnICA`'s case study on a large dataset of benchmarks, stemming from real-world code.

4.4 `Staticdeps`: dependency coverage *wrt.* `depsim`

In order to assess `staticdeps`' results, we implemented `depsim`, a simple dynamic analyzer instrumenting binaries using `valgrind`. It analyzes and reports memory dependencies of a given binary at runtime.

We use the binaries produced by `CesASMe` as a dataset, as we already assessed its relevance and contains enough benchmarks to be statistically meaningful. We also already have tooling and basic-block segmentation available for those benchmarks, making the analysis more convenient.

For each binary previously generated by `CesASMe`, we use its cached basic block splitting and occurrence count. Among each binary, we discard any basic block with fewer than 10 % of the occurrence count of the most-hit basic block as irrelevant.

For each of the considered binaries, we run our baseline dynamic analysis, `depsim`, and record its results.

For each of the considered basic blocks, we run `staticdeps`. We translate the detected dependencies back to original ELF addresses, and discard the Δk parameter, as our dynamic analysis does not report an equivalent parameter, but only a pair of program counters. Each of the dependencies reported by `depsim` whose source and destination addresses belong to the basic block considered are then classified as either detected or missed by `staticdeps`. Dynamically detected dependencies spanning across basic blocks are discarded, as `staticdeps` cannot detect them by construction.

Lifetime	cov_u (%)	cov_w (%)
∞	38.1 %	44.0 %
1024	57.6 %	58.2 %
512	56.4 %	63.5 %

Table 3. Unweighted and weighted coverage of `staticdeps` on `CesASMe`'s binaries

We consider two metrics: the unweighted dependencies coverage, cov_u, as well as the weighted dependencies coverage, cov_w:

$$\text{cov}_u = \frac{|\text{found}|}{|\text{found}| + |\text{missed}|} \qquad \text{cov}_w = \frac{\sum_{d \in \text{found}} \rho_d}{\sum_{d \in \text{found} \cup \text{missed}} \rho_d}$$

where ρ_d is the number of occurrences of the dependency d, dynamically detected by `depsim`.

These metrics are presented for the 3 500 binaries of `CesASMe` in the first data row of Table 3. The obtained coverage, of about 40 %, is lower than expected.

However, as we already pointed, dependencies matter only if the source and destination are not too far away. To this end, we introduce in `depsim` a notion of *dependency lifetime*. As we do not have access without a heavy runtime slowdown to elapsed cycles in `valgrind`, we define a *timestamp* as the number of instructions executed since beginning of the program's execution; we increment this count at each branch instruction to avoid excessive instrumentation slowdown. We re-run the previous experiments with lifetimes of respectively 1 024 and 512 instructions, which roughly corresponds to the order of magnitude of the size of a reorder buffer; results can also be found in Table 3. While the introduction of a 1 024 instructions lifetime greatly improves the coverage rates, both unweighted and weighted, further reducing this lifetime to 512 does not yield significant enhancements.

The final coverage results, with a rough 60 % detection rate, are reasonable and detect a significant proportion of dependencies; however, many are still not detected.

Dataset	Bencher	Datapoints	MAPE	Median	Q1	Q3	K_τ
Full	uiCA	3500	29.59 %	18.26 %	7.11 %	52.99 %	0.58
	+ staticdeps	3500	19.15 %	14.44 %	5.86 %	23.96 %	0.81
Pruned	uiCA	2388	18.42 %	11.96 %	5.42 %	23.32 %	0.80
	+ staticdeps	2388	18.77 %	12.18 %	5.31 %	23.55 %	0.80

Table 4. Evaluation through CesASMe of the integration of staticdeps to uiCA

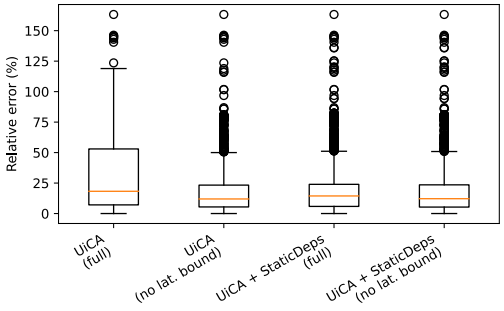

Figure 5. Statistical distribution of relative errors of uiCA, with and without staticdeps hints, with and without pruning latency bound through memory-carried dependencies rows

This may be explained by the limitations studied in subsection 3.6 above, and especially the inability of staticdeps to detect dependencies through aliasing pointers. This falls, more broadly, into the problem of lack of context that we expressed before: we expect that an analysis at the scale of the whole program, that would be able to integrate constraints stemming from outside the loop body, would capture many more dependencies.

4.5 Staticdeps: enriching uiCA's model

To estimate the real gain in performance debugging scenarios, however, we integrate staticdeps into uiCA.

There is, however, a discrepancy between the two tools: while staticdeps works at the assembly instruction level, uiCA works at the μOP level. In real hardware, dependencies indeed occur between μOPs; however, we are not aware of the existence of a μOP-level semantic description of the x86-64 ISA, which made this level of detail unsuitable for the staticdeps analysis.

We bridge this gap in a conservative way: whenever two instructions i_1, i_2 are found to be dependant, we add a dependency between each couple $\mu_1 \in i_1, \mu_2 \in i_2$. This approximation is thus pessimistic, and should predict execution times biased towards a slower computation kernel. A finer model, or a finer (conservative) filtering of which μOPs must be considered dependent — *eg.* a memory dependency can only come from a memory-related μOP — may enhance the accuracy of our integration.

We then evaluate our gains by running CesASMe's harness by running both uiCA and uiCA + staticdeps, as we did in Section 4.3, on two datasets: first, the full set of 3 500 binaries from CesASMe; then, the set of binaries pruned to exclude benchmarks heavily relying on memory-carried dependencies introduced in subsection 4.3. If staticdeps is beneficial to uiCA, we expect uiCA + staticdeps to yield significantly better results than uiCA alone on the first dataset. On the second dataset, however, staticdeps should provide no significant contribution, as the dataset was pruned to not exhibit significant memory-carried latency-boundness. We present these results in Table 4, as well as the corresponding box-plots in Figure 5.

The full dataset uiCA + staticdeps row is extremely close, on every metric, to the pruned, uiCA-only row. On this basis, we argue that staticdeps' addition to uiCA is very conclusive: the hints provided by staticdeps are sufficient to make uiCA's results as good on the full dataset as they were before on a dataset pruned of precisely the kind of dependencies we aim to detect. Furthermore, uiCA and uiCA + staticdeps' results on the pruned dataset are extremely close: this further supports the accuracy of staticdeps.

While the results obtained against depsim in subsection 4.4 above were reasonable, they were not excellent either, and showed that many kinds of dependencies were still missed by staticdeps. However, our evaluation on CesASMe by enriching uiCA shows that, at least on the workload considered, the dependencies that actually matter from a performance debugging point of view are properly found.

This, however, might not be true for other kinds of applications that would require a dependencies analysis.

5 Conclusion

In this article, we have presented the new heuristic-based analyzer, staticdeps, for statically identifying memory-carried dependencies of a binary basic block. We have shown that it can be used to beneficially extend state-of-the-art code analyzers, such as uiCA. In order to provide sound baselines for evaluating such extensions, we also have introduced the evaluation framework CesASMe, which is meant to compose measurement environments and code analyzers and make their results commensurable. In order to take into account the impact of a basic block's context on its measured throughput, the method we propose consists in reasoning at kernel-level by lifting the basic block-level predictions.

Finally, `CesASMe` has allowed us to exhibit a significant gain in accuracy, thanks to `staticdeps`, on latency-bound microbenchmarks generated from the HPC-native Polybench benchmark suite.

References

[1] Andreas Abel and Jan Reineke. 2019. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. *arXiv e-prints* abs/1911.03282 (2019). arXiv:1911.03282 http://arxiv.org/abs/1911.03282

[2] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[3] Andreas Abel and Jan Reineke. 2022. UiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 33, 14 pages. https://doi.org/10.1145/3524059.3532396

[4] AMD 2023. *AMD64 Architecture Programmer's Manual, volume 2*. AMD.

[5] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer International Publishing, 3–20. https://doi.org/10.1007/978-3-319-04519-1_1

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Technical Report OSU-CISRC-10/07-TR70. The Ohio State University.

[8] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sýkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 167–177. https://doi.org/10.1109/IISWC47752.2019.9042166

[9] Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. 2022. PALMED: Throughput Characterization for Superscalar Architectures. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 106–117. https://doi.org/10.1109/CGO53902.2022.9741289

[10] Fabian Gruber. 2019. *Performance Debugging Toolbox for Binaries: Sensitivity Analysis and Dependence Profiling*. Ph. D. Dissertation. Université Grenoble Alpes. http://www.theses.fr/2019GREAM071 2019GREAM071.

[11] Intel Corporation. [n. d.]. Intel Architecture Code Analyzer (IACA). https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

[12] Intel Corporation 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual, volume 1*. Intel Corporation.

[13] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[14] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. 2019. Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–6. https://doi.org/10.1109/PMBS49563.2019.00006

[15] Linux Kernel. [n. d.]. perf: Linux profiling with performance counters. http://perf.wiki.kernel.org/index.php/Main_Page.

[16] Charith Mendis, Saman P. Amarasinghe, and Michael Carbin. 2018. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. *CoRR* abs/1808.07412 (2018). arXiv:1808.07412 http://arxiv.org/abs/1808.07412

[17] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.* 89, 2 (2003), 44–66.

[18] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 428–439. https://doi.org/10.1109/HPCA.2014.6835952

[19] PoCC [n. d.]. PoCC, the Polyhedral Compiler Collection. https://www.cs.colostate.edu/~pouchet/software/pocc/.

[20] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBench/C: The polyhedral benchmark suite, version 4.2. http://polybench.sf.net.

[21] Nguyen Anh Quynh and the Capstone collaborators. [n. d.]. Capstone engine. https://www.capstone-engine.org/.

[22] Fabian Ritter and Sebastian Hack. 2022. AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 125 (oct 2022), 29 pages. https://doi.org/10.1145/3563288

[23] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. 2017. Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 423–435.

[24] Sony Corporation and LLVM Project. [n. d.]. LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html.

[25] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. https://doi.org/10.1147/rd.111.0025

[26] WikiChip. 2021. Intel Details Golden Cove: Next-Generation Big Core For Client and Server SoCs. https://fuse.wikichip.org/news/6111/intel-details-golden-cove-next-generation-big-core-for-client-and-server-socs/.