Simulating Wrong-Path Instructions in Decoupled Functional-First Simulation

Stijn Eyerman, Sam Van den Steen, Wim Heirman, Ibrahim Hur
Intel Corporation
{Stijn.Eyerman, Sam.Van.den.Steen, Wim.Heirman, Ibrahim.Hur}@intel.com

***Abstract*—Wrong-path speculative execution on an out-of-order processor core has no impact on an application's functionality and correctness, but it can impact performance by changing the state of caches and predictors. Not modeling wrong-path execution in performance simulation leads to performance projection errors up to 22% for our setup. However, wrong-path execution is challenging to model for common functional-first simulators, because the functional simulator is not aware of branch predictor misses and only provides correct-path instructions.**

We propose and evaluate multiple wrong-path modeling techniques for functional-first simulators, each with a different accuracy versus simulation speed balance. The novel instruction reconstruction with convergence exploitation technique proves to be the best balanced technique, with about 3× lower error than no wrong path modeling and about 2 to 3× faster simulation than full wrong path emulation.

I. Introduction

Processor performance simulation is an indispensable tool for architectural research, processor design, and to project performance in the context of procurement contracts. Due to the complexity of current processors and the tiny timescale, efficient *and* accurate processor simulation is a challenge, both in terms of implementation effort and simulation speed. Processor simulation can be divided into two tasks: functional and performance simulation. The goal of functional simulation is to construct the correct dynamic instruction stream and to extract data of each instruction that is needed by the performance simulator (instruction address, type, source and destination registers, data address, etc.). The performance simulator processes this data and simulates the performance of each instruction in each of the processor structures (cores, caches, network, memory, etc.).

Although other combinations theoretically exist, there are two common ways to implement a processor simulator: integrated functional and performance simulation, and decoupled functional-first simulation [2]. An integrated implementation emulates an instruction's functionality when it is in its actual execution stage in the performance model (execute-at-execute). Integrated simulation is more accurate than decoupled simulation, because it can adapt the instruction stream on micro-architectural events, such as branch predictor misses or load replays. Due to its execute-at-execute model, it also more accurately models the performance impact of memory operation synchronization. A drawback of integrated simulation is that it is less flexible: for example, the simulator knows that a branch is mispredicted only when it is actually executed and not yet at the fetch stage, so perfect branch prediction cannot be simulated.

Functional-first simulation has a separate functional simulator that runs ahead of the performance model, using a queue of instructions between the functional and performance simulator. Functional-first simulators are easier to implement and maintain, because they consist of two separate tools. They are also more flexible: you can reuse the same performance simulator for different functional simulators (e.g., different ISAs) or use the same functional simulator for different performance simulators (e.g., use an x86 emulator for an Intel architecture performance simulator and for an AMD architecture performance simulator). Furthermore, only a small fraction of the instruction opcodes in an ISA is used for most of the dynamic instructions of an application [3]. Therefore, not modeling the rarely used opcodes in the performance simulator does not impact accuracy much, while for a functional simulator, it is crucial to model all opcodes, because a single unmodeled instruction can lead to functionally incorrect execution or an application crash. Implementing the performance simulator independent of the functional simulator therefore does not require worrying about this completeness. Lastly, the decoupling of the functional and performance simulator enables them to run in parallel. An integrated simulator triggers instruction emulation one by one, leading to de facto sequential functional and performance simulation. This makes functional-first simulation faster than integrated simulation [2].

An important accuracy deficit of decoupled simulation is the inability to faithfully simulate wrong-path instructions, i.e., the instructions fetched after a branch misprediction until the misprediction is detected and the correct path restored. Although wrong-path instructions are flushed before they are retired and do not impact the architectural state, they can have an impact on performance, in particular through caches [24]. Wrong-path instructions access the instruction and data caches, thereby potentially bringing in new data and evicting older cache lines. If the newly inserted data is later used by a correct-path instruction, the correct-path instruction will take less time because the data is already cached. On the other hand, if the correct path accesses data evicted by the wrong-path instructions, its execution time will be longer because the data needs to be refetched to the cache.

Many processor simulators do not model wrong-path execution (see Section VI-A) because it complicates the simulator

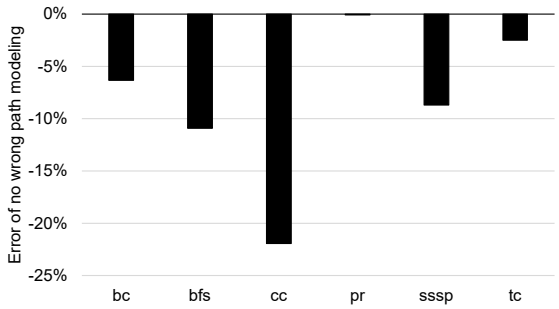


Fig. 1. Performance estimation error of no wrong path modeling for the GAP benchmarks.

implementation and because prior work indicates that wrong-path execution has not a large performance impact [11]. Branch predictors also have become highly accurate for commonly evaluated workloads, such as the SPEC benchmarks. However, we find that even for recent core configurations, wrong-path modeling can have a big impact on accuracy, and we expect this impact to increase for two reasons: First, emerging workloads, such as graph analysis [17], sparse neural networks [33] and graph neural networks [32], are more irregular than the established benchmarks. They have higher branch miss rates due to data-dependent branches and higher cache miss rates caused by sparse accesses. There is also a trend that regular dense workloads (such as HPC workloads and dense neural networks) are executed on GPUs and accelerators, leaving the irregular sparse applications as the main CPU workloads [6]. Secondly, high performance cores still trend towards increasing instruction depth and width (larger reorder buffers, more functional units), which increases the amount of speculative instructions. Furthermore, the increasing gap between core and memory speed leads to longer resolution times for mispredicted branches that depend on main memory accesses, increasing the time spent in wrong-path execution.

Figure 1 shows the performance estimation error of not modeling wrong path execution on a recent high-performance core configuration for the GAP benchmarks (see Section IV for our experimental setup). All errors are zero or negative (average -9.6% and up to -22%), meaning that not modeling the wrong path underestimates performance. This is because the GAP benchmarks show a high degree of convergence, where the wrong path starts prefetching data in the cache for the upcoming correct path code. Pagerank (pr) has no impact, because it has no conditional branches in its inner loop, and Triangle Count (tc) is mainly compute bound. In the results section, we also evaluate the SPEC CPU 2017 benchmarks. Some of them (mainly the regular FP benchmarks) are not impacted by wrong-path modeling, but a significant fraction (especially the INT benchmarks) also have negative errors, 2% on average and up to -9.7%.

In this paper, we discuss and evaluate three techniques to model the impact of wrong-path execution in a functional-first simulator. The first technique reconstructs the instructions along the wrong path and simulates the instruction cache, branch predictor and functional unit usage of these instructions. However, it cannot reconstruct the register contents along the wrong path, meaning that memory addresses that depend on register contents are unknown and that data cache accesses cannot be simulated. The second technique tries to also reconstruct register data—and thus memory addresses—, by using advanced features of the functional simulator, such as machine state checkpoints and instruction injection. This technique more accurately simulates the wrong-path performance impact, but it is much heavier to execute, reducing simulation speed with one to two orders of magnitude. The last technique uses the first technique for its low overhead, and tries to reconstruct memory addresses by detecting convergence between the wrong and correct path. Although this technique cannot reconstruct addresses if there is no convergence, it does cover the converging paths case, which has significant wrong-path performance impact.

The contributions of this paper are:

- Qualitative and quantitative discussion on the impact of not modeling wrong-path execution in functional-first simulators.
- Proposing three novel techniques to (partially) model the impact of wrong-path execution on performance in functional-first simulators.
- Evaluation of the three techniques, showing higher accuracy and limited simulation speed overhead for the wrong-path instruction recovery technique with memory address reconstruction using convergence detection: error reduces with a factor 3×, while simulation speed is 2× to 3× higher than the speed of the most accurate wrong-path emulation technique.

The next section discusses the characteristics of functional-first simulation. Next, we explain our proposed techniques to model wrong-path execution in a functional-first simulator. After explaining our experimental setup, we show simulation accuracy and speed results from our experiments. We conclude that the convergence exploitation technique considerably improves accuracy while not slowing down simulation speed as much as full wrong-path emulation.

II. Functional-First Simulation

Functional-first decoupled processor simulation consists of two separate engines: a functional simulator and a performance simulator. The functional simulator is responsible for correctly running an application and providing instruction details to the performance simulator. It can use dynamic binary instrumentation (e.g., Pin [22], DynamoRIO [10]), emulation (e.g., Qemu [8], Simics [23]), or a trace interpreter (for pre-recorded instruction traces). The performance simulator models the timing of the instructions in the processor pipelines, caches, memory, etc. It uses the instruction data from the functional simulator (instruction address, disassembled instruction, memory addresses) to accurately model the different components of a processor.

An important characteristic of functional-first simulation is that the functional simulation runs ahead of the performance

simulation. The functional simulator pushes instructions in a queue, which are consumed by the performance simulator. To ensure performance simulation is not delayed by the functional simulator, multiple instructions (tens up to thousands) are queued between the functional and performance simulator. This runahead can impact the accuracy of the performance model in multiple ways:

- Branch prediction is simulated in the performance simulator. When a branch is simulated and wrongly predicted, the instruction queue from the functional simulator contains the next correct-path instructions, and the functional simulator is already processing multiple instructions in the future. Wrong-path instruction execution can therefore not be modeled. Instead, the performance simulator halts instruction fetch until the branch is executed (in simulation time), after which correct-path fetch restarts (with some extra latency to model squashing instructions and restoring register rename state).
- For parallel applications with inter-thread synchronization, synchronization instructions are functionally executed before their performance simulation. The exact timing of synchronization instructions can impact the performance and/or the control flow of an application, for example in a work-stealing parallel application. Applications that are sensitive to synchronization instruction timings might therefore not be modeled accurately.

This paper handles the wrong-path simulation issue. The goal is to model the micro-architectural and corresponding performance impact of executing wrong-path instructions as accurately as possible in a functional-first simulator, in particular the state of instruction and data caches.

III. Wrong-Path Reconstruction in Functional-First Simulation

We developed three techniques for wrong-path reconstruction in functional-first simulation. The first technique reconstructs the instructions without emulating their functionality. This means that data-dependent information (e.g., memory addresses) is not reconstructed and cannot be used in the performance model (e.g., for modeling the data cache). The second technique performs fully functional emulation of the wrong path, including reconstructing the data and memory addresses. This is the most accurate method, but also the slowest (slowdown of 2× to 100×). The last technique uses the first technique as baseline, and tries to reconstruct memory addresses based on convergence between the wrong and correct path. It is almost as fast as the first method, while reconstructing most of the memory addresses in converging code, where wrong-path execution has the largest performance impact.

We describe our techniques based on the simulator we used to implement them: Sniper [12] with Intel Pin [22] as functional simulator. Most of the techniques can also be used on other functional-first simulators, but in particular the second wrong-path emulation technique uses specific Pin features, which are potentially not supported by other functional simulators.

A. Instruction Reconstruction using Code Cache

The instructions provided by the functional simulator are correct-path instructions only, because the functional simulator does not model the branch predictor. It is only when the branch predictor is simulated in the performance model that the branch misprediction is detected. The branch predictor model provides the wrong-path target: the next instruction if the branch is predicted not taken, the branch target if the branch is predicted taken, or the predicted target for an indirect branch. If a particular static branch instruction is executed multiple times (e.g., in a loop) and has multiple outcomes, the functional simulator has at some point provided instruction streams for several paths (taken and non-taken for a conditional path, multiple targets for an indirect branch). Therefore, we implement a code cache between the functional and performance simulator, keeping the information of past emulated instructions. This cache is indexed by the instruction address, and keeps the instruction decode information: instruction address, instruction type, input and output registers.

In case a branch is wrongly predicted, the performance model looks up the start of the wrong path in the code cache. If it is not present, it resorts to default branch misprediction modeling: halting fetch until the mispredicted branch has finished its execution stage. If it is present, the instruction is fetched from the code cache and sent through the simulated pipeline, along with the subsequent instructions. When a wrong-path branch is fetched, it is also predicted, and the predicted target is used to continue the wrong path. If at some point, the requested instruction is not in the code cache, the wrong-path reconstruction is stopped and fetch is halted until the resolution of the initial mispredicted branch.

Because we only reconstruct the instruction information and do not emulate the functionality of the reconstructed wrong-path instructions (calculations and memory data), we can only use data-independent information in performance simulation, including:

- the instruction address: for modeling instruction cache accesses,
- branch instruction type: for modeling branch prediction (in combination with the instruction address),
- instruction type: for modeling functional unit utilization and buffer occupation (ROB, load/store queue, etc.),
- input and output registers: for modeling dependences between instructions.

However, we cannot reconstruct data-dependent information, of which data memory addresses are most important to the performance model. In particular, we cannot model data cache and TLB accesses, load-store forwarding, address disambiguation, etc.

Data cache and TLB accesses typically have the largest impact on performance, because they change the state of the cache, which could have an impact on future correct-path instructions. The impact could be positive if wrong-

path instructions "pre-fetch" data in the cache for correct-path instructions, or negative if their cache accesses cause the eviction of useful data needed by the correct path. From our experiments (see Figure1), we found that the positive effect has the largest impact: wrong-path instructions eventually converge with the correct path and access caches with the correct memory address. When the correct-path instruction executes later on, it finds the data already in cache and it can proceed much quicker than when the wrong-path access is not modeled. Therefore, the next technique also aims to recover data addresses to correctly model this behavior.

B. Functional Wrong-Path Emulation

For accurate wrong-path simulation, the performance simulator needs to know the addresses of the data that the instructions access on the wrong path. This can only be achieved by actually emulating the wrong-path: simulating the calculations and loaded memory data of the wrong-path instructions. In a functional-first simulation model, the performance simulator cannot do functional emulation, so we need to implement this in the functional simulator. For Pin, the functional simulator we use in our setup, we start by taking a checkpoint of the current register state, to be able to resume execution after the branch miss is detected. Next, we use the 'PIN_ExecuteAt' method[1] to redirect execution to the wrong path and start emulating the wrong path. Stores, as well as exceptions, need to be suppressed since the functional state (memory and OS state) should not be affected by the wrong path. In a user-level functional simulator such as Pin, we need to end the wrong path on system calls, because kernel code cannot be instrumented. Store data could be kept in a separate structure to enable forwarding the data to future loads, but because the wrong path is usually short, store-to-load forwarding does not occur frequently. Once we are done executing down the wrong path, we restore the register checkpoint and continue along the correct path.

To decide when to head down the wrong path, the functional simulator contains a copy of the branch predictor model and initiates a list of wrong-path instructions when a misprediction is modeled. The wrong path is always followed for one reorder buffer (ROB) size worth of instructions (plus the frontend pipeline buffers). The timing simulator can then follow the wrong path until the initial branch miss is detected, and discard the unneeded instructions of the wrong path.

This technique models the impact of wrong-path execution most accurately, but it heavily reduces simulation speed by a factor of 5× and more. Taking a checkpoint, redirecting functional simulation, injecting instructions and restoring the checkpoint cause a big overhead on the functional simulator. It is also an implementation and debugging challenge to avoid application crashes, since many unexpected things can happen on a wrong path (including exceptions, invalid instructions, and other weirdness that a binary instrumentation tool such as Pin may not be prepared to handle). Lastly, the functional simulation frontend needs to support this feature. For example, a trace frontend cannot implement this, because the trace only contains correct-path instructions.

[1]https://software.intel.com/sites/landingpage/pintool/docs/98690/Pin/doc/html/group__CONTEXT.html

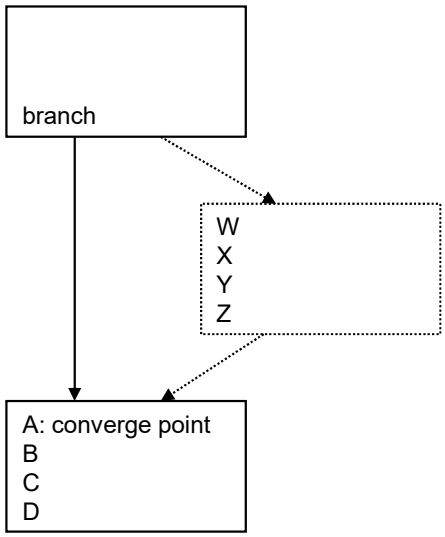


Fig. 2. One-sided branch; the 2 possible paths after the branch are WXYZ-ABCD and ABCD. Instruction A is the convergence point. In a one-sided branch, the convergence point is always the first instruction of one of both paths, which could be the correct or wrong path.

C. Wrong-Path Memory Address Reconstruction for Converging Code

The main conclusion of the two described techniques is that light-weight wrong-path simulation does not model the effect with the largest impact (data cache prefetching), while an accurate technique has too much overhead. Because wrong-path simulation has the highest impact for applications with converging code, this last technique targets wrong-path data cache access modeling for converging code. It exploits the fact that the functional model runs ahead of the performance model, so we can take a peek in the future correct-path instructions to find the memory addresses accessed by the converged wrong-path instructions. These addresses are used during wrong-path simulation to more faithfully simulate wrong-path data cache behavior.

This technique uses the code cache from the first technique. Additionally, it checks whether the wrong and correct path converge at some point, and whether the memory instructions after the convergence point are control and data independent of the instructions before the convergence point. The latter check is necessary because addresses may change when they depend on non-converged code. An important pitfall of this technique is to be overly optimistic: using addresses from correct-path instructions during wrong-path simulation will by construction cause a cache hit when the correct path instruction is simulated. Therefore, we should be cautious to only use these addresses when it is certain that the addresses match, i.e., when the instruction control flow converges and the memory

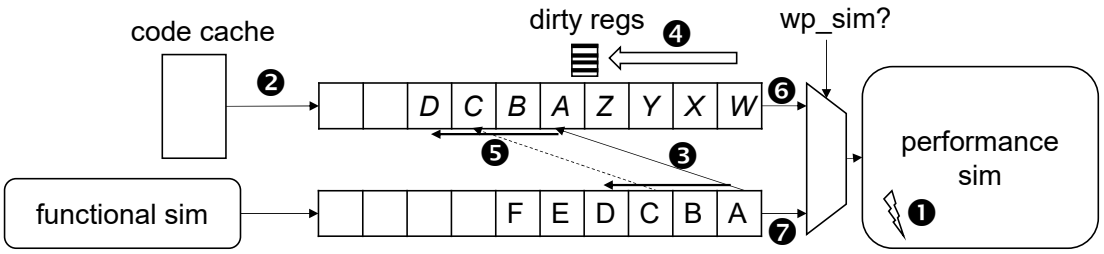


Fig. 3. Wrong path convergence detection and address reconstruction (instructions are the same as in Figure 2; ABCD is the correct path, WXYZABCD is the wrong path). When the performance model detects a branch miss (1), the code cache generates wrong-path instructions (in italic) (2). Next, it is checked whether the correct path or wrong path start converges (3) (in this example, the convergence point is the start of the correct path). Meanwhile, the registers written in the other path before the convergence point are collected (4). From the convergence point, both the correct and wrong path are scanned for convergence and data addresses are copied if the instructions match and are data-independent from the pre-convergence code (5). When this is finished, wrong-path simulation starts (6), until the performance model simulates the execution of the mispredicted branch and instruction fetch is steered back to the correct path (7).

instruction is control and data independent of non-converging code.

1) Convergence detection: On a branch miss, we detect convergence by going over the future wrong-path and correct-path instructions and by looking for matching instructions. Because of the optimism pitfall and because the wrong path is limited by the reorder buffer (ROB) size, we conservatively check for convergence. We only detect convergence for one-sided branches, see Figure 2. This means that we only check if the first instruction of the wrong path matches with an instruction at most ROB size instructions in the correct path *or* if the first correct path instruction matches with an instruction in the wrong path. In other words, if the branch is taken and wrongly predicted not taken, we check whether the fall-through path finally reaches the branch target, or reversely, if the correct fall-through path reaches the start of the wrong path (the branch target). As a result, we need to do at most 2 times ROB size comparisons, i.e., comparing the first wrong-path instructions with ROB size correct-path instructions, and comparing the first correct-path instructions with at most ROB size wrong-path instructions.

2) Independence check: If convergence is detected, we need to check which correct-path instructions after the convergence point are data independent of the wrong-path and correct-path instructions before the convergence point. Thereto, we build a list of written registers in the correct and wrong path before the convergence point. Next, we go over the correct path after the convergence point and check whether the instruction pointers match with the instructions along the wrong path after the convergence point. When we find a memory operation that matches with the wrong path and that is data-independent of the pre-convergence code (through register dependences), we copy the memory address to the wrong-path instruction information. This address is then later used in cache simulation when that wrong-path instruction models its memory access. Note that we only check for register dependences, not for through-memory dependences. Keeping track of memory dependences has a higher overhead, because you need to keep a list of all store addresses, compared to a limited register file size list for register dependences.

Figure 3 summarizes the steps taken by the simulator on a branch misprediction.

3) Overhead and limitations: Convergence-based wrong-path memory address recovery has an impact on converging code only and models only the positive prefetching impact of wrong-path data cache accesses. It also adds some overhead: at each branch miss, the future wrong-path and correct-path code is searched for convergence. If not enough instructions are in the queue between the functional and performance simulator, convergence checking can either be skipped or a message can be sent to the functional simulator to generate more instructions, adding time overhead. Nevertheless, the simulation speed is only slightly lower than that of the instruction reconstruction technique, and much higher than that of the functional wrong path emulation. We evaluate the simulation speed overhead of this technique in the results section.

We limit the convergence check cases to single-side branches (if-then, no if-then-else) which means we could miss convergence for some branches, leading to an underestimation of the performance gain. On the other hand, we do not check for through-memory dependences, which could lead to an overestimation. We made both choices to limit the overhead of convergence detection, and found that they already provide substantial accuracy gains for applications with converging code.

IV. Experimental Setup

We implemented the three wrong-path modeling techniques in an in-house version of Sniper [12]. We perform simulations with 4 simulator versions:

1) No wrong-path modeling (default)
2) Instruction reconstruction wrong path modeling
3) Wrong path modeling with instruction reconstruction and memory address reconstruction by exploiting convergence
4) Wrong path emulation

We want to evaluate each technique in how accurately it models the impact of wrong-path execution on performance. Because we don't have an integrated simulator that has the exact same performance model as our simulator—except for the

wrong-path modeling—to use as a reference, we assume that wrong path emulation (option 4) is the most accurate model. Therefore, we define error as the performance estimation difference with wrong path emulation. Note that functional wrong-path emulation is not always able to reconstruct all wrong-path instructions (e.g., it needs to stop on a system call, or the injection fails for some reason), so the actual error against fully correct wrong-path modeling might be different from the number we report. However, we assume the reported numbers are a good indication of the impact of wrong-path execution on performance and of how close the other techniques are to correctly modeling wrong-path execution. We also measure simulation speed to compare the four options.

As benchmarks, we use the GAP benchmark suite [7], composed of high-performance CPU implementations of graph analytics kernels. Graph analytics is used for big data semantic analysis by getting insights out of a large set of data and their relations. They impose important challenges to current processor architectures: very large data sets, data dependent branches and sparse irregular data accesses. The GAP benchmarks have characteristics that stress wrong-path modeling: high branch miss rate, high data cache miss rate and converging code. The latter is caused by the recurring structure of graph applications: the same function is applied to all vertices (or edges), and each function application is independent. After applying the function to one vertex, the code jumps to the next vertex, so if a branch miss occurs while processing the first vertex, code converges on the next vertex. Usually, the functions are relatively simple, so multiple function applications reside in the same reorder buffer, leading to convergence within a ROB size number of instructions.

The GAP benchmarks are heavily impacted by wrong-path modeling, and as such amplify the accuracy gain obtained by wrong-path modeling. They also have converging code, benefiting our convergence based technique. Applications that are insensitive to wrong-path modeling should not be impacted by our techniques, but applications that have a negative wrong-path impact and/or are not converging will only see accuracy gains for the unfeasibly slow wrong path emulation technique. To cover a larger application range, we also include results for the SPEC CPU 2017 benchmarks [1] (all SPECrate INT and FP benchmarks). To limit simulation time, we simulate a single 1 billion instruction sample per benchmark-input pair, gathered using the SimPoint method [26] (34 sampled traces in total).

Because wrong-path modeling is a core-internal model, we perform single-core simulations. We configure our simulator similar to a P-core of an Intel Alder Lake system (also known as Golden Cove microarchitecture) [27]. Table I shows the simulated core configuration parameters. We downscale the LLC and memory bandwidth to reflect the available LLC capacity and memory bandwidth per core in common SKUs.

TABLE I
SIMULATED PROCESSOR CONFIGURATION

Frequency	2.5 GHz
ROB size	512
dispatch width	6
commit width	8
L1 I-cache	32KB, ass 8
L1 D-cache	48 KB, ass 12
L2 cache	2 MB, ass 16
branch predictor	TAGE-SC
LLC size	1.9 MB
memory bandwidth	5.3 GB/s
memory latency	50 ns

V. ACCURACY AND SPEED EVALUATION

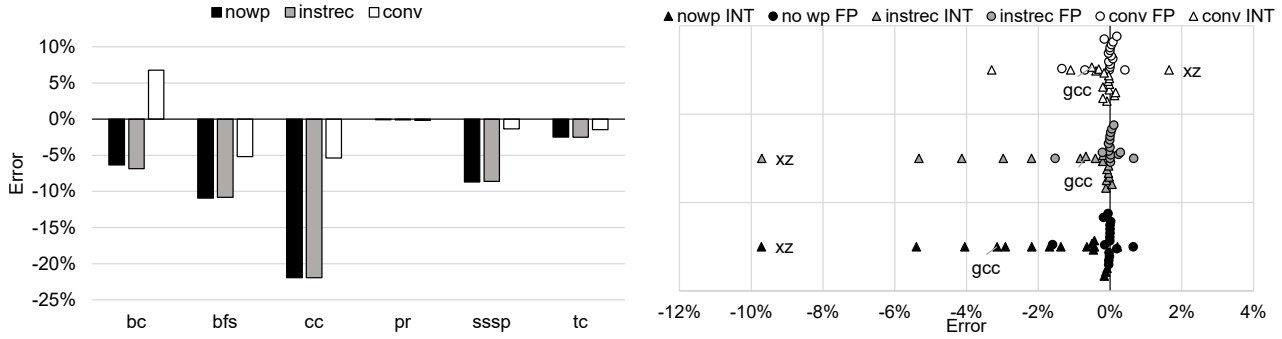
A. Accuracy of Approximate Wrong-Path Simulation

Wrong-path emulation is the most accurate simulation technique, but infeasibly slow for most practical simulation studies (up to 100x for very branch-miss-intensive applications). Therefore, we developed approximate wrong-path modeling techniques, namely wrong path instruction reconstruction and address reconstruction by exploiting convergence. Figure 4 shows the error of no wrong-path modeling (nowp), instruction reconstruction (instrec) and convergence exploitation (conv). Note that the nowp results are the same as in Figure 1.

For the GAP benchmarks, instruction reconstruction has a very small or no impact on error reduction. These applications have a low instruction memory footprint and are therefore not sensitive to instruction cache interference by wrong-path instructions. Convergence exploitation, on the other hand, significantly decreases the error for benchmarks that had a strong negative error. By detecting convergence and reconstructing memory addresses, we faithfully model the positive interference effect of converging wrong-path instructions prefetching data for the correct path. For Betweenness Centrality (bc), the error turns positive, meaning that part of the positive interference was offset by negative interference. Because the convergence exploitation technique only models positive interference, the negative interference effect now becomes visible. The average error is 9.6% for no wrong path modeling, 9.7% for instruction reconstruction and 3.8% for convergence exploitation.

The SPEC benchmarks are split into the integer (INT, triangles) and floating-point (FP, circles) benchmarks. The no wrong path results (bottom) show that most FP benchmarks are around 0% error, while the error distribution for the INT benchmarks is much larger. SPEC FP benchmarks mainly consist of regular number-crushing code with no hard-to-predict branches, while INT benchmarks are less regular and have data-dependent branches, explaining the larger impact of wrong-path modeling.

For the instruction reconstruction technique (middle), a few benchmarks, such as `gcc`, shift from negative towards 0% error. These benchmarks have a higher instruction cache miss rate than average, and wrong-path execution prefetches instructions into the instruction cache, leading to fewer correct-path instruction cache misses and therefore higher perfor-



(a) GAP benchmarks (b) SPEC benchmarks

Fig. 4. Error of wrong path modeling techniques for the GAP benchmarks (left) and SPEC benchmarks (right; distribution per technique; error ranges with many points have larger Y-value ranges). No wrong path model = nowp, instruction reconstruction = instrec, convergence exploitation = conv.

mance. However, the largest negative errors remain: 24% of the benchmarks still have a negative error ($< -0.5\%$), with 5 having an error below -2% (up to -9.7%).

The convergence exploitation technique (top) solves these outliers: only 15% remain with a negative error, with one outlier to -3.3%; 82% now have an error around 0% (was 64% with no wrong path modeling) and one (`xz`) has a positive error that is slightly higher than for no wrong path modeling. The increase in positive error is expected: the convergence technique targets positive interference and does not model negative interference. Benchmarks where negative and positive interference (partially) compensate for each other now show larger positive errors, with `xz` as the most pronounced example. However, the net accuracy gain is positive: more benchmarks are close to 0% and the average error decreases by a factor 4 for the SPEC INT benchmarks: from 1.97% (no wrong-path modeling) to 0.49% (convergence exploitation). The SPEC FP benchmarks' average error is 0.20% across all techniques.

In conclusion, benchmarks where wrong-path interference has a large impact, such as the GAP benchmarks, see a large accuracy improvement of using convergence exploiting wrong path modeling (average error reduces from 9.6% to 3.8%). A more diverse set of benchmarks with less pronounced wrong-path interference sensitivity move from a negatively skewed error distribution to a more narrow distribution around 0%, and an average error reducing from 1.97% to 0.49%.

B. Simulation Speed

One of the main benefits of functional-first simulation is its simulation speed, as the functional simulator and performance simulator run in parallel and are loosely synchronized (using the intermediate instruction buffer). All three wrong-path models slow down simulation because they need to simulate more instructions than the baseline no wrong path model. Instruction reconstruction is the most light-weight technique, followed by convergence exploitation, which is instruction reconstruction extended with convergence and independence checking. Wrong path emulation is the slowest technique due to the heavy-weight emulation in the functional simulator. In addition, it is also the hardest technique to implement.

TABLE II
FRACTION OF WRONG PATH INSTRUCTIONS RELATIVE TO CORRECT PATH INSTRUCTIONS.

	instrec	conv	wpemul
bc	240%	117%	69%
bfs	160%	86%	35%
cc	236%	53%	19%
pr	12%	12%	5%
sssp	166%	135%	66%
tc	50%	33%	18%

To evaluate simulation speed, we recorded the simulation time for all simulations, and normalized them to the no wrong path model, which is the fastest technique. For the SPEC benchmarks, which are mainly not or lightly branch miss heavy, the average slowdown for the instruction reconstruction technique is $1.12\times$ (up to $4.8\times$), the convergence exploitation technique is on average $1.13\times$ slower (up to $4.1\times$), and for the wrong path emulation, the slowdown is $2.1\times$ on average and up to $16.2\times$. The GAP benchmarks are very branch miss heavy, spending up to 75% of their execution time in wrong-path execution. Therefore, wrong path modeling has a much larger impact on simulation speed: an average $3.2\times$ slowdown for instruction reconstruction (up to $6.2\times$), $4.0\times$ on average for convergence exploitation (up to $5.1\times$) and a $13.1\times$ average slowdown for wrong path emulation, with an outlier of $157\times$.

The reconstruction techniques do not touch the functional simulator, so their slowdown is caused by the performance simulator (reconstructing wrong path and simulating wrong-path instructions). The slowdown of the wrong path emulation technique is caused by the functional simulator, because the performance simulation model is the same as for the other two techniques and both simulators execute in parallel.

We conclude that the convergence exploitation technique provides the best accuracy-simulation speed balance of the proposed wrong-path modeling techniques: it is only slightly slower than instruction reconstruction, while significantly reducing the error for applications that are sensitive to wrong-path modeling.

TABLE III
LOW-LEVEL METRICS FOR THE CONVERGENCE EXPLOITATION TECHNIQUE. FRACTION OF BRANCH MISSES WHERE CONVERGENCE IS FOUND (CONV FRAC); AVERAGE NUMBER OF INSTRUCTIONS UNTIL THE CONVERGENCE POINT (CONV DIST); FRACTION OF WRONG PATH MEMORY OPERATIONS WHOSE ADDRESSES ARE RECOVERED USING CONVERGENCE EXPLOITATION (ADDR RECOVER); FRACTION OF WRONG-PATH L2 MISSES THAT ARE COVERED BY THE CONVERGENCE EXPLOITATION TECHNIQUE VERSUS THE WRONG PATH EMULATION TECHNIQUE (WP L2 MISS).

	Conv frac	Conv dist	Addr recover	WP L2 miss
bc	98%	30.1	34%	44%
bfs	93%	13.7	34%	24%
cc	92%	6.9	44%	73%
pr	75%	24.9	31%	0%
sssp	62%	28.4	31%	32%
tc	90%	7.6	54%	9%

C. In-Depth Analysis

To gain further insight into the different wrong-path modeling techniques, we recorded different metrics during simulation. Table II shows the fraction of wrong-path instructions executed by the wrong-path models for the GAP benchmarks. It is relative to the correct path instruction count, so 100% means that there are as much wrong-path instructions executed as there are correct-path instructions. The high numbers indicate the importance of wrong-path modeling for the GAP benchmarks, with up to 2.4× more wrong-path instructions than correct-path instructions. Pagerank (pr) is an exception because it has no data-dependent conditional branch in its inner loop.

If we compare the different techniques, it might be counter-intuitive that instruction reconstruction (instrec) executes more wrong-path instructions than convergence exploitation (conv), which in its turn executes more wrong-path instructions than wrong-path emulation (wpemul). Instruction reconstruction does not model memory accesses, so each memory operation is modeled as a cache hit. That means that the wrong-path proceeds faster compared to accurately modeling cache misses and their latency, as done by the wrong-path emulation and (partially) by the convergence exploitation technique. Therefore, during the branch miss resolution time, i.e., the time it takes to execute the dependence path to the branch and the branch itself, more instructions can be executed in the instruction reconstruction model than in the other models. The convergence exploitation technique models the memory operations that can be reconstructed accurately, including their latency to access caches and memory. Therefore, it executes fewer wrong-path instructions than the instruction reconstruction technique, but more than the (most accurate) wrong-path emulation model, because not all memory addresses can be recovered. These numbers again show that convergence exploitation is closer to correct wrong-path modeling than instruction reconstruction, with a limited simulation speed overhead.

Table III shows further details on the convergence exploitation technique for the GAP benchmarks. The first column is the fraction of branch misses where convergence is detected, i.e., the start of the correct or wrong path is found later on in the other path. Again, the high numbers show that GAP benchmarks have a lot of convergence, with up to 98% of the branches converging. This is because their inner loop typically goes over the neighbors of a specific vertex and applies the same function on all of them independently. If there is a branch misprediction in that function, the execution will eventually jump to the next iteration of that loop, providing a convergence point.

An additional factor contributing to the impact of convergence is that the functions applied to a vertex' neighbors are relatively simple and thus short in terms of instruction count. That is shown in the second column, which indicates the average number of instructions between the branch and the convergence point (of the path where the first instruction is not the convergence point). It shows that after 7 to 30 instructions, the paths converge and the remaining instructions in the ROB (up to 512) are all potentially the same, providing a large opportunity to prefetch correct data to the caches. Of course, mispredictions could also occur along the wrong path, making both paths diverge again, but the numbers show the large potential for convergence.

The address recover column shows the fraction of memory instructions along the wrong path for which we are able to recover the addresses by exploiting convergence. This fraction is much lower than the fraction of branches where convergence is found. Memory operations that depend on the non-converging code are not recovered, as well as memory operations that lay on a path that diverges further after the convergence point, because of a misprediction along the wrong path. So the deeper in the wrong path, the less likely can the memory address be recovered. However, it is more important to recover the addresses close to the branch miss, because these will have the most impact on cache hits. Memory operations deeper in the wrong path are less crucial. Triangle count (tc) has remarkably more addresses recovered. This benchmark has a low cache miss rate, and the branches do not depend on instructions that have a cache miss, so its resolution time and depth of wrong path is much lower than for the other applications. The closer a memory operation is to the mispredicted branch, the higher the probability that it can be recovered, explaining the higher recover fraction.

The last metric compares the amount of misses in the L2 cache along the wrong path between the convergence exploitation technique and wrong path emulation (baseline). This number is more important than the amount of addresses that are recovered, because these misses will change cache content and prefetch data for the correct path. For most benchmarks, a large fraction of misses is recovered, especially those that see a large performance swing for the convergence exploitation technique. Pagerank spends few time in wrong path and triangle count has fewer cache misses in the wrong path, explaining their low number. Note that the overall cache miss rate does not change significantly across the techniques: no or only few additional misses are introduced during wrong-path simulation, the converging misses along the wrong path are turning correct-path misses into hits.

VI. Related Work

A. Functional-First versus Integrated Simulation

As discussed in the introduction, the two most common types of processor simulators are decoupled functional-first and integrated (often called timing-directed) simulation. Examples of functional-first simulators are Sniper [12], COTSon [5], MARSS [25], CMPSim [20] and ChampSim [19]. These simulators do not model wrong-path execution by default. ASIM [16] and gem5 [9] are integrated simulators, where the instruction function is emulated exactly when it is executed in the timing simulation model (execute-at-execute). An in-between form is execute-at-fetch [21]: the instructions are emulated when they are fetched in the timing model. This is similar to functional-first, but without letting the functional simulator run ahead to the timing model. This makes it easier to redirect the instruction stream on a branch misprediction (which is detected at the fetch stage), but it reduces simulation speed because the functional and performance simulator now run in lockstep, heavily reducing the potential of parallel simulation. The abundance of functional-first simulators compared to integrated simulators is an indication that they are easier to implement. Akram and Sawalha [2] composed a list of processor simulation tools, with most timing simulators using the functional-first technique. They also mention the performance benefit of functional-first simulators compared to integrated simulators.

To speed up simulation, several papers propose FPGA-accelerated performance simulation [14], [31]. Because an FPGA-based functional simulator is complex to implement, FAST [14] executes it on the CPU. The functional model sends instruction data to the FPGA model, adopting the functional-first simulation model. To model wrong-path execution, the functional simulator is able to roll back to the mispredicted branch and emulate wrong-path instructions. This is similar to our wrong-path functional emulation technique. The slowdown of this technique versus not modeling wrong path is not reported. RAMP [31] integrates both the functional and performance model on an FPGA, but they model in-order cores without branch prediction, which have no speculative wrong-path execution.

B. Importance of Wrong-Path Simulation

The performance impact of wrong-path execution has been recognized before. While Cain et al. [11] claim that wrong-path execution has a negligible impact on performance, Mutlu et al. [24] report up to 10% error when not modeling wrong-path execution in simulation. The main difference between both studies is the memory latency: Cain et al. assume 70 cycles, while Mutlu et al. model at least 250 cycles. If a branch misprediction depends on a cache miss (and thus memory access), the memory access latency determines the branch resolution time and therefore the time spent in wrong-path execution. We find that wrong-path modeling has indeed a low impact for most of the commonly used SPEC benchmarks, but it can have a much larger impact for other emerging applications, such as graph analysis.

Sendag et al. [29], [30] find that in a multicore processor, wrong-path cache accesses can have an even larger impact by interfering in the cache coherence policy, leading to more coherence traffic, state changes and writebacks. They also propose a technique to reduce the impact of wrong-path execution. We have only evaluated single core execution, but our wrong-path simulation techniques also apply to multicore simulation.

Chandra et al. [13] show that wrong-path execution has an even larger impact on power consumption than on performance. They propose an offline trained model to predict wrong-path power consumption during trace-based simulation, which cannot simulate wrong-path execution. They do not model the performance impact of wrong-path execution.

C. Exploiting Convergence

Our convergence exploitation technique uses convergence between the wrong and correct path to reconstruct memory addresses without functionally emulating the wrong path. Several researchers point to the potential of this convergence to improve the performance of an out-of-order processor: converging control- and data-independent instructions on the wrong path are not flushed and refetched, only the instructions that truly depend on the mispredicted branch are flushed. Rotenberg and Smith [28] were the first to describe this mechanism for trace processors. Collins et al. [15] describe how convergence can be detected in hardware and describe two techniques to exploit convergence to improve performance. Al Zawawi et al. [4] propose a novel ROB-less design, based on re-execution buffers to execute control and data dependent instructions after a mispredicted branch. Eyerman et al. [18] use hint instructions to guide the processor to flush only branch-miss dependent instructions.

VII. Conclusions

Wrong-path instruction execution can have a big impact on performance, especially when wrong-path instructions eventually converge with correct-path instructions and start fetching useful data to the caches. However, simulating the wrong path in a functional-first simulator—a common simulation paradigm with high simulation speed and flexibility—is challenging, because the functional simulator only simulates correct-path instructions. We propose and explore multiple techniques to model wrong-path instruction execution in a functional first simulator, from instruction reconstruction using a code cache to full-blown wrong path emulation. In particular, we propose a novel technique that looks for converging code in the correct and wrong path, which enables reconstructing memory addresses along the wrong path, crucial for modeling the impact of wrong-path execution on cache state. We find that this technique forms an interesting balance between accuracy and simulation speed: it reduces error from 9.6% to 3.8% for the branch-miss-heavy GAP benchmarks and from 1.97% to 0.49% for the more diverse SPEC INT benchmarks. At the same time, it is 1.9 to 3.5× faster than wrong-path emulation.

REFERENCES

[1] "SPEC CPU 2017." [Online]. Available: https://www.spec.org/cpu2017

[2] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *Ieee Access*, vol. 7, pp. 78 120–78 145, 2019.

[3] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter, "X86-64 instruction usage among C/C++ applications," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, pp. 68–79.

[4] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary, "Transparent control independence (TCI)," in *34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 448–459.

[5] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[6] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the role of the CPU in the era of CPU-GPU integration," *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012.

[7] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[8] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[10] D. Bruening and T. Garnett, "Building dynamic instrumentation tools with dynamorio," in *Proc. Int. Conf. IEEE/ACM Code Generation and Optimi zation (CGO), Shen Zhen, China*, 2013.

[11] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, "Precise and accurate processor simulation," in *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, vol. 8, 2002.

[12] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[13] S. Chandra, R. Jayaseelan, and R. Bhargava, "Speculative path power estimation using trace-driven simulations during high-level design phase," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 630–637.

[14] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 249–261.

[15] J. D. Collins, D. M. Tullsen, and H. Wang, "Control flow optimization via dynamic reconvergence prediction," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 129–140.

[16] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert *et al.*, "Asim: A performance model framework," *Computer*, vol. 35, no. 2, pp. 68–76, 2002.

[17] S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 282–292.

[18] S. Eyerman, W. Heirman, S. Van Den Steen, and I. Hur, "Enabling branch-mispredict level parallelism by selectively flushing instructions," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021, pp. 767—-778.

[19] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[20] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[21] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 53–64.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[23] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[24] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1556–1571, 2005.

[25] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 1050–1055.

[26] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[27] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger *et al.*, "Intel alder lake cpu architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.

[28] E. Rotenberg and J. Smith, "Control independence in trace processors," in *32nd ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Nov 1999, pp. 4–15.

[29] R. Sendag, A. Yilmazer, J. Y. Joshua, and A. K. Uht, "The impact of wrong-path memory references in cache-coherent multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1256–1269, 2007.

[30] R. Sendag, A. Yilmazer, J. J. Yi, and A. K. Uht, "Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, p. 10.

[31] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "Ramp gold: an fpga-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 463–468.

[32] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[33] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *51st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 15–28.