Modeling Architectural Support for Tightly-Coupled Accelerators

David J. Schlais, Heng Zhuo, Mikko H. Lipasti
University of Wisconsin-Madison
schlais2@wisc.edu, hzhuo2@wisc.edu, mikko@engr.wisc.edu

***Abstract*—As proposed accelerators target finer-grained chunks of computation and data movement, it becomes increasingly important to couple them tightly with the processor, avoiding long invocation delays. However, the large implementation design space of these Tightly-Coupled Accelerators (TCAs) makes it difficult to balance trade-offs between hardware complexity and accelerator performance. Previous performance models for accelerators focused on the penalties associated with loosely-coupled accelerators, which abstracted away many of the fine-grained interactions with complex out-of-order structures and program behaviors that have large impacts on TCA performance. In this paper, we introduce an analytical model that studies TCA behavior when interacting with the core, in the context of both high and low memory bandwidth applications supporting various levels of speculative and out of order (OoO) execution. Our analytical model reduces the turnaround time in early design stages when estimating performance gains over detailed simulation with tolerable error. We also discuss potential design choices that can impede the benefits that come with TCAs, and illuminate differences with traditional accelerators.**

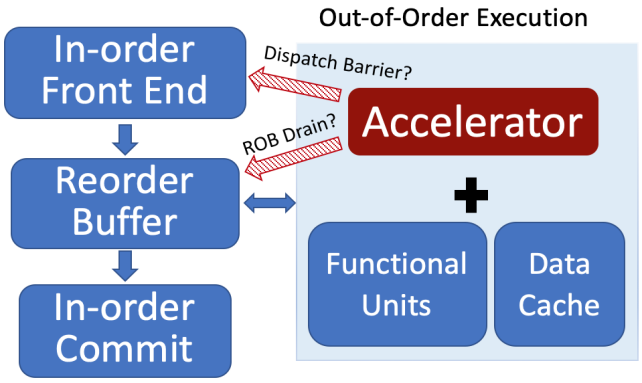


Fig. 1: High-level system diagram of a tightly-coupled accelerator (TCA) integrated into an OoO core. Our studies evaluate how requiring dispatch barriers or reorder buffer (ROB) drains sometime has significant performance impacts.

I. Introduction

When defining an instruction set architecture (ISA) for processors, there is often a balance between creating more complicated, specialized instructions for code density and/or performance, and simpler instructions for generality. Commonly repeated software routines can evolve over time from software functions to designated hardware units. For example, floating point operations used to be calculated using floating point software emulation. When workloads increasingly used floating point operations, architects tightly integrated specific hardware units to replace the slower software functions. A more recent example is how the increase of matrix-matrix multiplication on GPUs has led architects to create designated 4x4 matrix-matrix multiplication instructions and dedicated hardware in the Nvidia Volta architecture called 'tensor cores.' Jia et al. [1] analyze how closely these tensor cores approach optimal theoretical throughput for matrix multiplication. Similarly, SIMD extensions (e.g. [2]) can be viewed as the same class of specialized hardware, where previously software-defined loops of operations are replaced with designated wide registers and operators.

Additionally, computer architects introduced hardware accelerators to specialize very large algorithms and operations to deliver expected performance improvements without violating power budgets. These accelerators improve energy efficiency over a processor's general-purpose pipeline and increase performance by trading off software flexibility for performance. Initial hardware accelerators often targeted very large tasks, such as video encoding [3], graphics, etc. Then, loosely-coupled *sea of accelerators* [4] became a model to accelerate several different domains, which is now a widely adopted approach in commercial products, such as mobile phones.

Naturally, as designers search for additional acceleration opportunities, they necessarily consider increasingly fine-grained software tasks. For this reason, we see tightly-coupled accelerator (TCA) proposals for heap management, hash maps, string functions, and other fine-grained tasks [5] [6]. This general trend has moved towards accelerating smaller algorithms and groups of instructions, and is increasingly moving into the realm of specialized functional units, similar to complex instructions in a CISC (complex instruction set computer) ISA.

Prior work has identified key principles that drive the increasing popularity of specialized hardware accelerators. Nowatzki et al. [7] explains in detail the benefits that come from accelerators, as well as the characteristics of tasks that can greatly benefit from acceleration. Although their discussion focused on coarse-grained accelerators, the same principles are relevant for fine-grained acceleration.

This paper is a first attempt to address these principles, design choices, and trade-offs in the context of TCAs. For the rest of this paper, we define a TCA to be an accelerator that is a hardware block that replaces a software function/routine, is invoked via a dedicated ISA instruction, reserves an entry

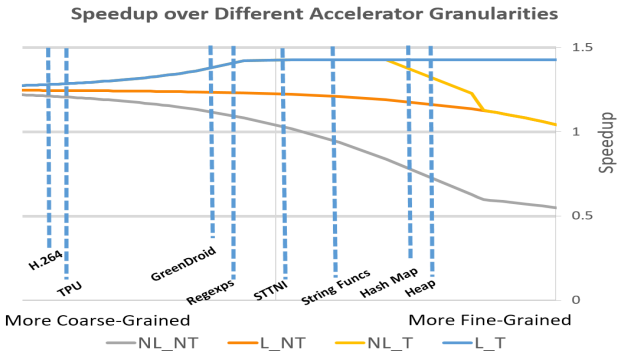


Fig. 2: High-level analysis of program speedup for accelerators invoked at different granularities using the novel analytical model proposed in this paper. Markers for the granularity of H.264 [3], Google's TPU [8], GreenDroid [9], speech recognition using STTNI [10], heap management [6] [5], regular expression [6], string function [6], and hash map [6] acceleration are estimated for points of reference. We insert parameters into the model based on an ARM A72 processor, assume 30% of code acceleratable, with an accelerator speedup of 3x. Note that the TCA mode choice of full (L_T), partial (L_NT and NT_L), or no (NL_NT) support for concurrency between the accelerator and program execution has a larger impact on program speedup in **fine-grained** accelerators. These 4 modes are explained in detail in Section III.

in the reorder buffer (ROB), has in-order commit semantics, and integrates with the processor's register file and/or coherent memory hierarchy. Fig. 1 shows a high-level diagram of an example TCA integrated into an OoO core.

II. Motivation

Prior TCA proposals often provide little discussion of how these accelerators should be integrated into cores, leaving a large design space to explore, with large differences in power, area, and even performance. For example, the single decision on whether or not the accelerator is allowed to execute speculatively has trade-offs in performance, (by allowing the core to take advantage of ILP around the accelerator), area, and power. The performance increase provided by speculative execution requires dedicated hardware in charge of rollback on misspeculation, as well as register and memory dependency resolution mechanisms.

One can imagine that when accelerating small chunks of code (fine-grained acceleration), failing to support speculation by forcing the processor to drain its ROB and/or stall dispatch every time the accelerator is invoked can significantly impact performance. The analytical model proposed in this paper captures this expected result, as shown in Fig. 2. For very coarse-grained accelerators, OoO support is less important for speedup. For moderate granularity, full OoO support delivers better speedup than coarse accelerators, since the ROB experiences fewer stalls due to the long latency of the accelerator, exposing more ILP for non-accelerated instructions. For fine-grained scenarios, inadequate OoO support (NL_NT) can lead to slowdowns. This result motivates development of analytical models that provide insight into these effects. Creating a high level model (discussed in detail in Section III) allows the designer to numerically estimate these impacts and make informed design estimations as a first step prior to the laborious development of a detailed simulator.

Prior accelerators focused on coarse-grained tasks, where pipeline drains and fills created negligible impact on overall speedup (see left-hand side of Fig. 2). For this reason, prior work that models accelerator speedup such as LogCA [11] focused on coarse-grained accelerators. The negligible performance impacts from pipeline drains and fills were naturally ignored in the model. LogCA also assumed that the CPU was idle during accelerator execution, which similarly targets coarse-grained accelerators. The Gables Model for SoC accelerators [12] focuses mostly on hardware resource utilization, and estimating an accelerator's overall speedup given the architecture's configuration, memory bandwidth available to the accelerator, and computational complexity of the work done by the accelerator per byte of data. This is a useful step in order to estimate an accelerator's speedup and could be used in early design stages or even in conjunction with our analytical model.

We show that at the granularity of many *current* accelerators proposed in academic papers, different implementations have a dramatic range of overall program speedup, where some implementations can actually cause slowdown. Combining this fact with the trend towards accelerating smaller, more fine-grained tasks motivates the need for being able to accurately estimate the differences in hardware implementations for various levels of speculative execution. Further impacts of the ROB/OoO integration are discussed in subsequent sections.

Understanding the trade-offs between each implementation will be an important step for choosing the optimal implementation based on the type of accelerator, processor architecture, program behavior, and desired metrics. However, to our knowledge, no prior research has numerically evaluated these trade-offs, leaving architects to either try to intuitively predict the optimal implementation or spend a large amount of time to design and test each potential implementation. We create an analytical model to quantify these impacts. After validating the model over various workloads and accelerators, we discuss overarching trends in TCA design, and summarize key conclusions. The novel contributions to this work are as follows:

- Creating a first-order analytical model for estimating the performance of fine-grained accelerators with and without support for out-of-order (OoO) execution of leading and/or trailing instructions (speculation and/or no dispatch barrier)
- Validating the analytical model and comparing errors of one synthetic and two realistic accelerators compared to a cycle-accurate simulator.
- Describing design space considerations, takeaways, overarching trends, and conclusions from the analytical model for architects to consider in building TCAs.

III. Analytical Model

When designing a TCA, the architect must decide whether or not the accelerator is allowed to reorder its execution with leading (L) and/or trailing (T) instructions. If the accelerator is allowed to execute concurrently with L instructions, this means that the accelerator is executing speculatively, since L may contain unresolved branches or instructions that will raise an exception. If the accelerator is allowed to execute concurrently with T instructions, trailing instructions must know whether or not they are dependent on the TCA's output(s) through dependency resolution hardware. The result of allowing concurrency with L and T instructions is improved performance, but the downside is the complications of designing hardware to ensure program correctness. Papers have discussed the performance benefits that can come from tightly-coupled accelerators [6] [5], and it is often assumed that these accelerators allow full out of order execution with respect to the rest of the program. Specifically, speedup is usually estimated by replacing the time spent within an acceleratable region with the accelerator execution time, which assumes the core can maintain its OoO execution rate around the accelerator (or in other words, similar to having both L and T concurrency enabled). In this section, we describe the hardware design changes needed to operate in different modes of OoO execution, and the equations our analytical model uses to predict performance when operating in each mode. The full list of input parameters used in are analytical model are shown in Table I.

Previous work from [13] shows how to create a mechanistic first-order model for program performance based on interval analysis. Specifically, performance is estimated by counting events for branch mispredictions, ICache misses, TLB misses, short/long DCache misses, etc. After adding penalties associated for those events, based on the processor/workload characteristics, one can estimate the overall performance. One of the key insights is: OoO performance modeling can be simplified by looking at the throughput of useful instructions dispatched in the *(in-order)* front end of the processor. Since the TCAs described in this paper are invoked by inserting an instruction through the normal pipeline of an OoO core, one can view accelerator invocations as a new category of 'special events' that occur during program execution. Depending on the implementation of the TCA, it may act like a branch misprediction (zero useful dispatches until the accelerator instruction commits), typical long-latency instruction, or new type of event whose penalty is related to the window drain and/or execution time of the TCA.

Our analytical model estimates overall performance changes using interval analysis of either an entire program or region of interest. For the rest of the paper, we discuss this in the context of analyzing an entire program, but the process is identical for a specific region of interest. Invocation frequency (v) is calculated by dividing the number of accelerator invocations by the number of total instructions. This variable represents the rate of acceleratable tasks in the baseline program. Regardless how these invocations are distributed throughout execution, the model predicts speedup assuming an even distribution of accelerator invocations. The model also assumes that that IPC of the core for non-acceleratable instructions is equal to the average program IPC before acceleration, and speculative accelerator instructions can begin execution as soon as they are dispatched. Note that this assumption may be overly optimistic for workloads with many dependencies between the acceleratable and non-acceleratable instructions. The average dispatch rate is also assumed to be equal to the IPC while the core is not stalled, and 0 when the core is stalled. The model also estimates the average non-acceleratable work to be evenly distributed across accelerator invocations.

variable	name
a	% acceleratable code
v	invocation frequency
IPC	Instrctions / cycle
A	Acceleration factor
s_{ROB}	size of ROB
w_{issue}	issue width
t_{commit}	commit stall

TABLE I: Analytical model parameters

When integrating an accelerator into an OoO core, the simplest hardware design eliminates concurrent execution of instructions with the accelerator, while fully supporting OoO execution will lead to the most complicated design. Partially allowing OoO execution will fall somewhere in between. Fig. 3 shows an overview of the four different degrees of integration we consider in this work.

The baseline assumes a full software implementation with no invocations to the accelerator. The time can be estimated by the number of instructions in the interval divided by the average program IPC, as shown in (1).

$$t_{baseline} = \frac{\#inst}{IPC} = \frac{1}{v * IPC} \tag{1}$$

Accelerator execution time can either be an explicitly provided latency inserted by the architect, or estimated by the number of 'accelerated_instructions' divided by the accelerator effective IPC (A*IPC). This comes out to be:

$$t_{accl} = \frac{\#accl\_inst}{A * IPC} \frac{a}{v * A * IPC}. \tag{2}$$

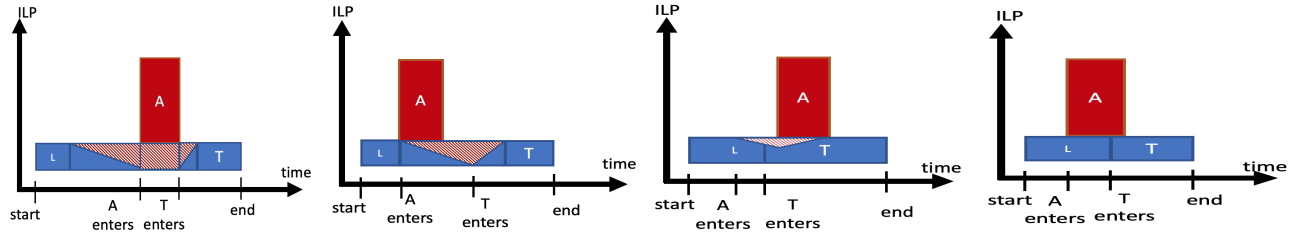
Similarly, we can estimate the execution time of non-accelerated instructions in the core by taking the total number of non-accelerated instructions and dividing it by the program's average IPC as shown in (3).

$$t_{non\_accl} = \frac{1 - a}{v * IPC} \tag{3}$$

The rest of this section describes penalties specific to the four different TCA implementations and how they are added into our analytical model.

A. Non-Leading & Non-Trailing (NL_NT)

We call this mode Non-Leading & Non-Trailing (NL_NT), since this mode does **not** (N) allow TCA execution to overlap with leading (L) instructions or trailing (T) instructions.



(a) NL_NT (b) L_NT (c) NL_T (d) L_T

Fig. 3: Example of effective ILP in the execute stage for a core with a TCA operating in four different modes. This figure shows an interval with leading (L) instructions, trailing (T) instructions, and a single accelerator (A) instruction. The striped sections show reduced ILP in the core caused by TCA mode. Interval analysis allows overall program execution to be estimated by the execution of the average interval.

Specifically, a NL_NT TCA must wait until all leading instructions commit, and all trailing instructions must stall in the front end until the TCA commits. This simplifies the TCA hardware design, in that the designer knows the processor will never squash an in-flight TCA instruction. It does not have to checkpoint internal states, since it will never have to roll back. Similarly, no dependency checking hardware is required in the accelerator, since the accelerator is never executing simultaneously with other instructions.

For performance modeling purposes, as soon as the accelerator is dispatched into the issue queue, the dispatch stage will drop to zero useful instructions dispatched per cycle until the ROB window drains as shown in Fig. 3. From the front end perspective, this causes a penalty for 'window_drain' (t_{drain}) time, plus the pipeline backend time to commit instructions (t_{commit}). Drain time is based on the latency of the critical path in a given ROB window size. Work from Eyerman et al. [13] shows a power-law relation between window size and critical path length for SPEC2006 benchmarks. Our model allows the user to manually adjust the window drain time if it is explicitly known for the target program, but by default will estimate t_{drain} by knowing the program IPC and using the power-law relationship to estimate the average critical path length for the given ROB size. If t_{non_accl} is smaller than t_{drain}, denoting shorter instruction execution in the core than the estimated drain time, t_{non_accl} is used instead.

After the window has drained, the accelerator begins execution, but the dispatch rate of trailing (T) instructions will still remain zero until the accelerator executes (t_{accl} cycles) and commits (for a second t_{commit} penalty). This second barrier further simplifies the TCA hardware design, in that the designer does not need to implement memory or register renaming for accelerator outputs to eliminate false dependencies in OoO execution of younger instructions. It also does not require any data forwarding or dependency checks with respect to the trailing (T) instructions. Adding these fill penalties of length t_{accl} and t_{commit} with the drain penalties of t_{drain} and t_{commit} and results in a total NL_NT execution time of:

$$t_{NL\_NT} = t_{non\_accl} + t_{accl} + t_{drain} + 2 * t_{commit}. \quad (4)$$

B. Leading & Non-Trailing (L_NT)

We call this mode Leading & Non-Trailing (L_NT), since this mode allows TCA execution to overlap with leading instructions, but no trailing instructions can be dispatched until the TCA commits. Designing in this mode means that due to the very nature of speculative execution, it is possible that the processor squashes the TCA instruction during or after execution due to a branch misprediction or other similar event. The accelerator architect must guarantee any state changed by the accelerator will be reverted to its previous state if squashed.

Because trailing instructions cannot execute concurrently with the TCA, the front end stalls, dispatching no useful instructions until the TCA commits. From the front end perspective, no useful instructions are dispatched for $t_{accl} + t_{commit}$ cycles. Adding this penalty to the baseline gives us the total execution of the (L_NT) mode shown in (5).

$$t_{L\_NT} = t_{non\_accl} + t_{accl} + t_{commit} \quad (5)$$

C. Non-Leading & Trailing (NL_T)

We call this mode Non-Leading & Trailing (NL_T), since this mode does not allow TCA execution to overlap with leading instructions but does allow the trailing instructions to be dispatched immediately after the TCA instruction, even as it waits for younger instructions to drain. Hence, non-dependent trailing instructions can execute before the TCA finishes, or even begins execution. This requires the architect to guarantee dependence checks for registers and memory locations that the TCA will modify. This dependency check can be done through modifications of existing structures such as the load/store queue (LSQ) and register renaming and forwarding logic.

Our model predicts that the front end can continue dispatching useful instructions until the ROB (potentially) fills up during accelerator execution. The front end can issue w_{issue} instructions per cycle until the ROB becomes full. The accelerator will have a delayed execution start of t_{drain} while the ROB drains. If $t_{accl} + t_{drain} + t_{commit}$ is greater than the ROB fill time ($t_{ROB\_fill} = \frac{s_{ROB}}{w_{issue}}$), then the front end will stall for every cycle after that until the TCA instruction commits. The time that the core will stall in this NL mode can then be estimated using (6).

$$t_{NL\_ROB\_full} = MAX(0, t_{drain} + t_{accl} + t_{commit} - t_{ROB\_fill}) \quad (6)$$

The total execution in the NL_T mode will be the greater of the accelerator's (delayed) execution time or the core execution time with its penalties.

$$t_{NL\_T} = MAX(t_{non\_accl} + t_{NL\_ROB\_full}, \; t_{accl} + t_{drain} + t_{commit}) \quad (7)$$

D. Leading & Trailing (L_T)

We call this mode Leading & Trailing (L_T), as it allows both the TCA to execute speculatively, as well as trailing instructions to be dispatched while the TCA is executing. This mode will have the best performance, since under normal operation, the dispatch stage can continually dispatch useful instructions. In order to prevent any dispatch stalls, the core/TCA must be able to roll back on misspeculation, and the control logic must enforce register and memory dependences correctly between the TCA and both L and T instructions. From the front end perspective, there is a continuous dispatch of instructions and there are no penalties unless the accelerator execution time is long enough to fill the entire ROB. However, unlike the case in NL_T, the accelerator begins execution immediately, and is not delayed for t_{drain} cycles. Therefore, the ROB full time is slightly different:

$$t_{ROB\_full} = MAX(0, t_{accl} - t_{ROB\_fill}). \quad (8)$$

In this mode, the CPU's traditional execute units will have the highest utilization while the TCA is executing. The total execution time becomes:

$$t_{L\_T} = MAX(t_{non\_accl} + t_{ROB\_full}, t_{accl}). \quad (9)$$

E. Bringing it all together

Analytical models that include variables and terms that are easily understood can be helpful from an intuitive standpoint. They can provide important general understanding by enabling limit studies and analysis of theoretical upper/lower bounds. They also can help provide early-stage design guidance and performance estimates without running detailed simulations. The key factors identified above regarding support for overlapping TCA execution with leading and/or trailing instructions can dramatically affect the overall speedup, but speedup is also highly dependent on the characteristics of the individual accelerator. We use the analytical model to show all estimated speedups for all four models.

The model attempts to capture the duration of front end stalls dispatching into the ROB based on architecture, accelerator, and program properties for each of the four levels of speculation. By incorporating these TCA overheads in a mechanistic program model of the CPU front end similar to [13], we use interval analysis to estimate program speedup. On average, the CPU issues roughly IPC useful instructions per cycle, but can drop to 0 when the TCA forces a ROB drain or ROB fill penalty as described in Sections III-A through III-D and summarized below. The overall speedup can then be estimated by comparing the execution time and penalties in all four accelerator modes to the baseline implementation.

- Reorder buffer drain (NL modes): The modes requiring the ROB drain will have an accelerator dispatch stall time equal to the window drain time. The window drain time can be explicitly entered into the formula or estimated based on ROB window size and program behavior as described in [13]. However, if T instructions are allowed to issue OoO, the core front end does not see a drain penalty until after the ROB is full of T instructions before the accelerator commits.
- Reorder buffer fill (NT modes): The duration of the fill penalty for modes that do not allow trailing instructions to execute concurrently with the accelerator is associated with the duration of the accelerator execution time. Early in the design cycle, this accelerator latency can be estimated, or it can be exact if the accelerator design is already well defined. Until the accelerator is done dispatching, executing, and committing, no further instructions can be dispatched.

IV. Methodology

To validate our analytical model, we use the cycle-accurate simulator gem5 [14]. We test our model over a synthetic microbenchmark to measure error over many different accelerator/workload scenarios, followed by experiments with heap management and matrix multiplication microbenchmarks to test real-world proposed accelerators that have both low (heap accelerator) and high (matrix accelerator) memory bandwidth requirements.

We replace acceleratable code of the compiled benchmarks with a special accelerator instruction. The NL drain and NT dispatch penalties are modeled by setting flags in gem5 for the accelerator, either to be a non-speculative instruction or to serialize the pipeline and not allow younger instructions to be dispatched until commit. Accelerator instructions issue all memory requests needed required to execute, assuming ability to issue contiguous loads for sizes up to 64B (same width as an AVX-512 register). An accelerator instruction is not considered committed until all memory and compute (micro)operations of the accelerator have committed.

Our synthetic heap microbenchmarks do not contain dependencies between the acceleratable and non-acceleratable code, and the matrix-multiplication only has dependencies from memory loads/stores. As long as the processor has a large enough window, we assume in the analytical model that the core can continue executing useful instructions while dependencies are resolved (unless the ROB fills). The accelerator is assumed to have its own compute resources and does not need to arbitrate for functional units in the core. However, all memory requests required by the accelerator pass through arbitration for shared access to the core's LSQ and memory hierarchy. Priority is granted based on age (program order).

We first test our model with a sweep of microbenchmarks which varies over many different invocation frequencies and percentage of acceleratable code to validate over many different scenarios. The accelerator instructions (or acceleratable code in the baseline case) are randomly distributed within the program to see how our model performs while violating our assumption of uniform TCA distribution. Each point in Fig. 4 represents a separate workload instance with a differing number of acceleratable instructions replaced by accelerator invocations. This provides us several different workload scenarios to compare our analytical model against. After gaining confidence in the model over a variety of scenarios and accelerator designs, we then focus on both heap management and matrix-matrix multiplication accelerators to test more realistic TCAs.

Malloc and free calls in the TCMalloc library were evaluated to take about 39 and 20 cycles (69 and 37 x86 uops), respectively [15]. The proposed heap manager accelerator has single-cycle latency, which we assume in our model as well. Our heap microbenchmarks randomly perform malloc and free calls throughout the benchmark for the desired percentage of acceleratable code under the constraint that the heap accelerator will always have a pointer to return on malloc calls and always have an available entry for free calls (the common case).

Matrix-matrix multiplication accelerators are currently available in some of today's cores. NVIDIA's Tesla and Turing GPUs have tensor cores capable of computing 4x4 half-precision matrix-matrix multiplication. The Tesla-based Volta as described in [16] shows the major throughput improvements that this TCA can provide. They perform 4x4 matrix-matrix multiplications through the register file, with 4x4 FP16 input registers. Limited information is given on whether loading the 4x4 input registers happens through separate (vector) load instructions with coalescing, if the entire matrix is loaded and stored as 4x4 sub-matrices, or loaded through other means. In order to support arbitrary 4x4 matrix loads and operations, we take an educated guess that these registers are filled through gather operations of matrix load instructions, while the resulting product matrices are written back to memory via scatter operations.

Our tested matrix multiplication accelerators act similarly but with a few differences. Instead of operating through the register file, our implementation of matrix multiplication operates through memory loads and stores, since current CPUs do not have dedicated 4x4 matrix registers. This allows us to simulate matrix multiplication acceleration without making significant ISA and microarchitectural changes. By making the requests through memory rather than registers, we can easily adjust our instruction to be able to accelerate 2x2, 4x4, and 8x8 matrix multiplication sizes with minor architectural changes in the core apart from the TCA itself. This allows us to test our model over various accelerator designs.

However, one downside to this approach is that Volta's implementation of register-based acceleration will allow easier reuse of the output matrix for accumulation, while our implementation of outputs require redundant loads and stores through memory. Additionally, our implementation has separate load instructions for each matrix row, as opposed to the register-based separate sub-matrix storage which can be loaded in a single instruction. However, Volta's registers appear to be filled through multiple loads prior to invoking the matrix multiply [16], incurring a similar load latency to ours.

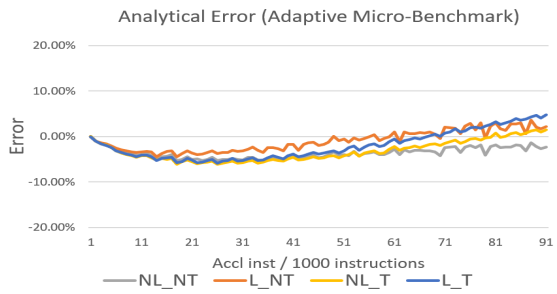


Fig. 4: Error of our analytical model speedup prediction compared to gem5 simulation of a synthetic microbenchmark while varying the number of accelerator instructions (increasing invocation freq and % acceleratable code).

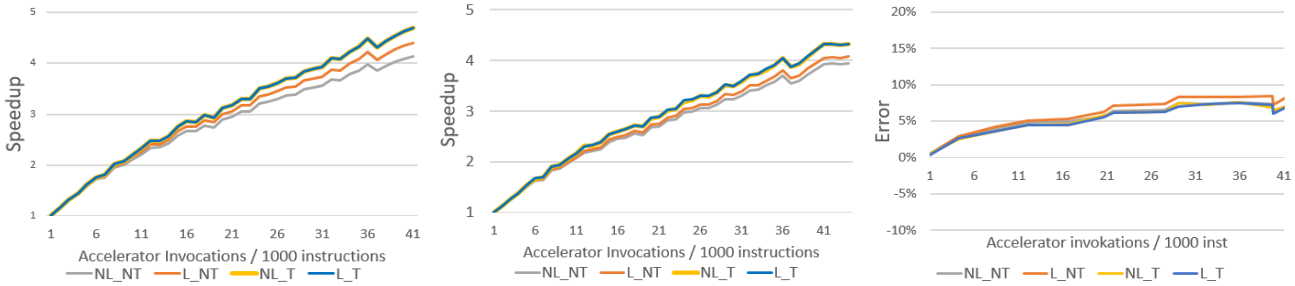
V. Model Validation/Verification

A. Adaptive Microbenchmark

We first validate our model through an adaptive microbenchmark that varies program and accelerator parameters. As we increase the number of accelerator instructions, we are increasing both the invocation frequency and the percent of acceleratable code. By randomly placing the accelerator instructions within the baseline microbenchmark, the compiler creates slightly different optimizations, slightly varying the program's base IPC in each resulting microbenchmark. This also helps compare the simulated versus analytical speedup when the benchmark does not match the model's first-order assumption of evenly distributed accelerator instructions. We end up testing over a wide variety of program behaviors with both low and high frequency of accelerator invocation and determine that our analytical model typically has less than 5% error (see Fig. 4), giving us confidence to test real-world applications as well.

B. Heap Manager TCA

We then applied this same method to current accelerator proposals, starting with the heap management accelerator. This accelerator contains hardware tables to store a subset of the free lists tracked by the the TCMalloc library, and provides accelerated (single-cycle) calls to both malloc and free. We built a microbenchmark that allocated from one of 4 different class sizes (0-32B, 33-64B, 65-96B, or 97-128B). The baseline executes the TCMalloc operations by invoking software library calls to the heap manager. For the accelerator, since the overwhelming majority of requests hit in the accelerator, we assume that the accelerator will never have to fall back to the software subroutine. By inserting



(a) Analytical model speedup (b) Gem5 simulation speedup (c) Error of analytical model

Fig. 5: Error of our analytical model in predicting accelerator speedup of heap microbenchmarks with different frequency of malloc/free function calls, each with 4 different TCA implementations. The NL_T line closely follows L_T. The highest error occurs at high invocation frequency, but still remains within a 10% error for a vast sweep of TCA invocation frequencies.

accelerator instructions in place of these function calls, we see increasing speedup as the frequency of malloc and free requests increase. Although our model has slightly higher error at higher invocation frequencies (up to 8.5% as seen in Fig. 5), we can still observe the same general trends of the effect that each of the 4 TCA modes has on overall program execution. This shows that our model still correctly predicts overarching trends for this workload.

C. Matrix-matrix multiplication TCAs

Since many machine learning and artificial intelligence applications involve matrix-matrix multiplication, we decided to further test our analytical model with different hardware accelerators with different size matrix-matrix operations. We created a software harness to calculate a 512x512 double-precision matrix-matrix multiplication. A naive implementation would do the entire multiplication in one triply-nested loop. However, since the two input matrices and output matrix have a total memory footprint of 6MB, much larger than the L1 D-cache (typically on the order of 32kB), there would be significant thrashing in the L1 cache that would limit temporal reuse. To optimize the algorithm to a L1 D-cache of 32kB, we calculate the 512x512 double-precision floating point matrix multiplication through 32x32 sub-matrix blocks at a time. The two input and output sub-matrices now only require 24kB, and all accesses to elements of these matrices should hit in the L1 D-cache after the first access. We can calculate a 32x32 output sub-matrix partial product by multiplying the two input matrices together. The partial product will continue accumulating by multiplying the next 32x32 sub-matrices in the larger 512x512 input matrices. After 16 32x32 sub-matrix multiplications, the first 32x32 sub-matrix in the 512x512 output matrix will be fully calculated. Our performance results are based on the time it takes to calculate the the full 512x512 output matrix.

Within the 32x32 matrix multiplication operations, we implemented an element-wise software kernel (our baseline), as well as accelerators that can directly multiply-accumulate 2x2, 4x4, and 8x8 sub-matrices. These accelerators all request memory addresses for the cache lines that they will need to access to do their computation. Within each of the accelerators, we also model the 4 different TCA implementations (NL_NT, NL_T, L_NT, and L_T).

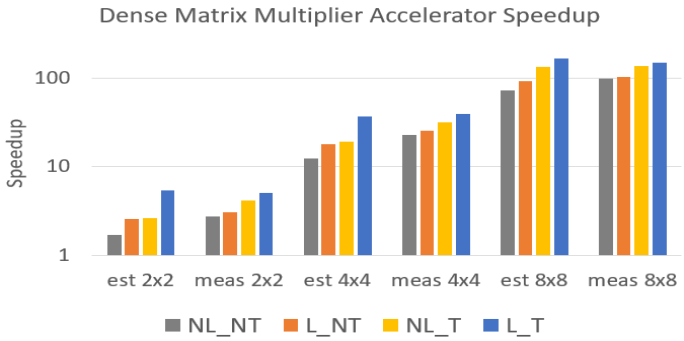
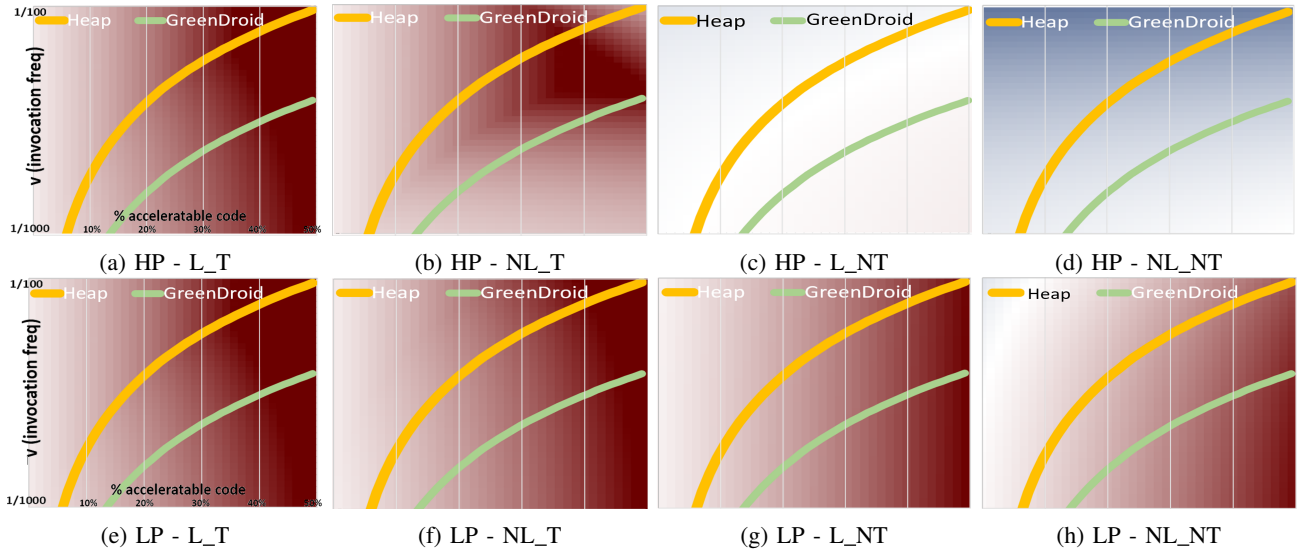


Fig. 6: Acceleration of a 512x512 dense matrix multiplication through 32x32 sub-matrix blocking. Speedup is shown on a logarithmic scale. 'Meas' denotes measured speedups of gem5 simulation and 'Est' denotes our analytical model estimated speedup, both relative to execution time of a software element-wise multiplication. Note the same general shape of the measured vs estimated speedup of the 4 different TCA implementations for the 2x2, 4x4, and 8x8 DGEMM accelerators.

As we can see from Fig. 6, our analytical model captures the general trends for all 4 TCA implementations. However, we see that the analytical model is slightly pessimistic for the non-L_T modes. Our error reaches as high as 44%. Although such errors seem rather high, they are being amplified by the very large speedups in these scenarios, and our model still accurately predicts the right relative trends, enabling valuable insights to accelerator design without the need for highly accurate absolute speedups.

As we consider the overall execution time of the 3 different accelerator designs and 4 different TCA implementations, note that there is a larger absolute difference in execution time between the 4 different modes of the 2x2 accelerator than the 4x4 and 8x8 accelerators. This is because the overall program speedup is much smaller in the 2x2 TCA case, increasing the absolute penalty. By the time we get to the 4x4 and 8x8 sub-



(a) HP - L_T (b) HP - NL_T (c) HP - L_NT (d) HP - NL_NT

(e) LP - L_T (f) LP - NL_T (g) LP - L_NT (h) LP - NL_NT

Fig. 7: Heatmap of speedups (red) and slowdown (blue) when sweeping over percent acceleratable code and invocation frequency (logarithmic scale). We map the locations heap manager and GreenDroid (GD) accelerators would fall over workloads with different % acceleratable code. The 2 rows from top to bottom show a mid-high performance (HP) OoO core, and low performance (LP) OoO core. The 4 columns from left to right are the TCAs operating in modes L_T, NL_T, L_NT, NL_NT.

matrix multiply-accumulate cases, the larger speedup from the accelerator amortizes the cost of the drain and fill penalties.

VI. Analytical Model Use Case

We want to use our analytical model to gain valuable insights into both existing and previously proposed fine-grained accelerators. In this section, we apply our model to the heap manager accelerator and to GreenDroid functions. GreenDroid [9] proposes mapping common functions in Mobile SOC workloads to TCAs with shared access to the L1 D-cache. This provides a use case of relatively fine-grained acceleration (hundreds of instructions) where we can use our analytical model to gain further insights.

The GreenDroid work shows the percentage of dynamic code execution that is run inside the given function, as well as the static number of instructions in that accelerator invocation. However, dynamic code execution increases both when (a) the function contains loops, increasing the number of dynamic instructions for each invocation, and (b) when the function is called more frequently. If the function has no loops and executes straight through, the function will have the largest invocation frequency. However, if the functions iterate many times per invocation, the accelerator will be called less often. In this analysis, we consider only the 9 functions described in [9] and assume straight-through execution of the functions for the highest invocation frequency case.

We apply our first-order analytical model varying 2 parameters (invocation frequency and percent acceleratable code) to create a 2D heatmap shown in Fig. 7. Red locations represent program speedup, blue areas represent program slowdown, and darker shades represent higher magnitudes of speedup/slowdown. We test over each of the 4 modes, as well as using processor characteristics from both a high performance core (1.8 IPC, 256 entry ROB, 4-issue) and low performance core (0.5 IPC, 64 entry ROB, 2-issue). We use the power-law analysis from [13] to estimate critical path instruction length from ROB size. We use the fixed-function accelerators from the heap manager, as well as estimate locations of GreenDroid functions and map where they lie on the 2D map. Each point on the curve represents the invocation frequency required to achieve a certain percent of acceleratable code. For example, a fixed-function accelerator will have to be invoked more frequently to obtain greater coverage (percent acceleratable code). Since Greendroid is motivated by energy efficiency rather than performance, we assume a much lower acceleration factor of 1.5x.

The curve for the heap accelerator is shown for reference of a more fine-grained accelerator than the GreenDroid functions. Using the analysis from our model, we make the following observations:

1) High performance core vs. Low performance core: The comparison between the first row and second row of Fig. 7 tells us that high performance cores are more sensitive to different modes of TCA. The first reason is that fixed-function accelerators will have higher relative acceleration factors on low performance cores due to the slower baseline execution time. Additionally, penalties arising from not allowing OoO execution are relatively higher on a high performance core due to the larger ROB size and drain penalties. Thus, architects should pay close attention to OoO integration when designing TCAs for high performance cores. For low performance cores, the impact on OoO integration is less severe.

2) Fine-grained vs. less fine-grained: Fine-grained accelerators need to be careful about slowdown, especially for NT

modes. Although both accelerators are fine-grained compared to conventional accelerators, Greendroid is less fine-grained than the heap manager. Consequently, for GreenDroid acceleration, the TCA implementation mode is less important, as the plots never cross into the slowdown range for these input parameters into our model. However, the heap manager is fine-grained enough that if it only achieved an acceleration factor of 1.5x in a HP core, it experiences slowdown when operating in the NT modes.

3) Limits of the model: Because it is a first-order analytical model, many details are omitted and these have the potential to mislead a designer. For example, from comparing L_T and NL_T mode, one may draw conclusion to not implement the L_T design, since it leads to a more complicated hardware design with little performance gain. Here, the penalties from non-speculative execution of TCA instruction are visually similar, since the core can usually cover the overheads with other OoO execution. This may be because of our assumption made in Section IV that IPC remains constant when the core is not stalled, regardless of dependencies. When there are many dependencies between the core and accelerator, the accelerator has a longer latency before it can begin execution, and core IPC will likely drop until the dependencies are satisfied when the accelerator completes execution. This could create a significant difference in practical use depending on the actual accelerator and workload. When using malloc/free, explicit dependencies between instructions will lead to noticeable slowdown in the absence of speculation support for TCA invocations, e.g. when younger instructions wait for a malloc pointer. This is a shortcoming of our simple, fast, and easy first-order model, and represents a very typical trade-off between a model's level of abstraction and its ability to capture every aspect of execution. Our model *generally* makes a reasonable prediction, but it is important to understand scenarios where it may fall short.

VII. Discussion

Accelerators typically have two primary functions: to increase performance and/or increase energy efficiency over the general-purpose core. For the case of the primarily energy efficiency-motivated accelerators, the overall speedup may not seem important. However, even in this case, our model can detect the areas in which non-L_T modes start creating program slowdowns. It is important for these accelerator designers to make sure these points are avoided for both performance and energy reasons. Program slowdown requires the core to run longer, increasing the amount of static energy consumed by the core, eroding the energy gains created by the accelerator.

For accelerators motivated by speedup, our analytical model quantifies costs associated with partial or full elimination of OoO execution. From this model, we can see that the higher the invocation frequency of the accelerator and the smaller the region of acceleratable code, the worse the relative penalty that arises from not supporting full OoO execution.

We also learn from our model that high-performance cores have the largest discrepancies between the 4 different modes.

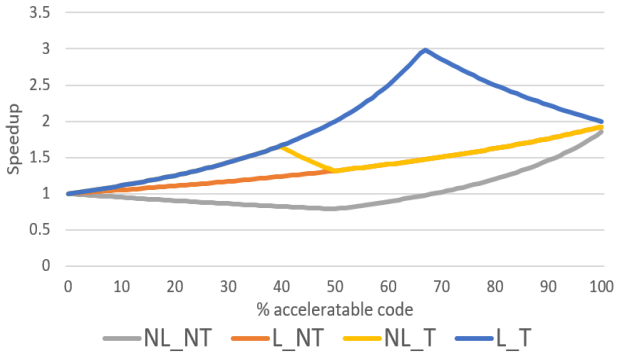


Fig. 8: Analytical model predicted speedup for a TCA of 100 instructions with a speedup factor of 2. Note that the maximum speedup does not occur at 100% acceleratable code, since concurrency exists between the core and TCA in OoO modes.

This is exacerbated for several reasons. First, the drain penalties are larger because ROB size is bigger. Second, the pipeline depth is longer in high performance cores, further increasing the drain and fill penalties. Lastly, barrier penalties are relatively higher because a fixed-latency accelerator has less relative acceleration in a high performance core than for a low performance core, making an effectively longer barrier penalty. Similarly, the overall speedup factor is higher for low performance cores. This means that designing accelerators for low-performance cores, perhaps for energy purposes, designers may decide to forgo accelerator complexity of L_T implementation with little overall impact on performance. This reduction of complexity will also further decrease power consumption of the accelerator.

We also learn from the analytical model that very coarse-grained acceleration and workloads with a high percentage of acceleratable code are less sensitive to the execution mode. Once the duration of the acceleratable code is on par with the latency of the longest instruction dependency chain in the processor window, the non-L_T modes begin having more substantial differences across the four modes. In the case of large portions of work being offloaded, such as the 8x8 matrix multiplication, the overall penalties of non-speculation are limited. Smaller portions of work that are offloaded, such as 2x2 matrix multiplication and heap manager accelerators have increased performance sensitivity and motivate tightly-coupled OoO execution.

Our model also helps us come to an interesting conclusion about a new form of concurrency that arises from supporting full OoO execution (L_T mode) for TCAs. We can see from Fig. 8 that a TCA with an acceleration factor of $A = 2$ can give the program a speedup of 3. OoO execution allows concurrency between the core and accelerator executing at the same time. This means that the maximum obtainable speedup when introducing a TCA is not just A, but actually $A + 1$. The peak overall speedup of 3 occurs when 67% of code is acceleratable. This is because for an accelerator with $A = 2$, work is evenly distributed between the TCA and the core when

the accelerator has 2x more work to execute than the core. For $A = 5$, the peak occurs when 5x more work is offloaded to the accelerator, or $\frac{5}{6}$ of the workload. Beyond this point, offloading more work to the accelerator diminishes performance, as the core becomes underutilized. Our model also shows maximum concurrency cannot be obtained when introducing dispatch stalls, non-speculative accelerator execution, and/or when the ROB fills up. As a side note, even in these cases, our model can still estimate where the highest possible concurrency is reached for the given input parameters. We also see interesting behavior in the NL_T mode, where a speedup local maximum is reached at a lower % acceleratable code, since the concurrency is maximized when the TCA latency plus delayed start (t_{drain}) is equal to core execution time. After this point, the TCA execution time plus delay determines overall speedup. At first, offloading more work to the TCA slows down execution due to work imbalance, but eventually the global maximum is reached once enough code is offloaded from the core to reduce the core's ROB drain time (t_{drain} approaches 0 as the program approaches 100% acceleratable code).

VIII. Future Work

In this work, we have discussed the performance evaluation for TCAs. However, each of the implementations require different hardware, each with various area and power costs. In order to create a more complete evaluation, a pareto-optimal curve of design implementations could show the trade-off between hardware costs, performance, and which (if any) design implementations fall outside of the curve and should not be considered. This paper assumes the ability to integrate with existing core structures such as the register file and memory cache hierarchy. Further microarchitectural analysis would be beneficial in determining the specific implementation details required to integrate various TCAs of different input/output requirements into core structures such as the register file and memory cache hierarchy.

Since each TCA has slightly different requirements, in order to easily integrate new TCAs into the same core, it may be beneficial to define a standard interface between each element of the core. This could allow more application-specific cores with faster turnaround and lower costs and effort towards processor design and validation.

On a different note, since misspeculation can cause additional delay penalties on the core, and branch mispredictions are the most common cause of misspeculation, partial TCA speculation (such as only speculating when on an instruction path with high-confidence outstanding branches) could also be a possible implementation for a design somewhere between the L and NL modes described in this work.

IX. Conclusion

In this paper, we describe and validate a model that estimates TCA performance with various levels of OoO execution. The model shows that for increasingly fine-grained accelerators, allowing OoO execution around the accelerator can have significant impacts on overall performance. Through this model, we also evaluate likely candidates for TCAs, and calculate different speedups based on implementation. Even with very different workloads, ranging from high memory and low memory applications, as well as high invocation frequency, the analytical model seems to show reasonable error and allow a designer to make general conclusions about specific accelerators. Through this, we show robustness in our analytical model.

X. Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by the National Science Foundation under grants CCF-1615014, CCF-1628384, and CCF-1813434.

References

[1] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[2] Intel, *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*, April 2019.

[3] Y.-W. Huang, B.-Y. Hsieh, T.-C. Chen, and L.-G. Chen, "Analysis, fast algorithm, and vlsi architecture design for h. 264/avc intra frame coder," *IEEE Transactions on Circuits and systems for Video Technology*, vol. 15, no. 3, pp. 378–401, 2005.

[4] Y.-T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich cmps: From concept to real hardware," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 169–176, IEEE, 2013.

[5] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating memory allocation," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 33–45, ACM, 2017.

[6] D. Gope, D. J. Schlais, and M. H. Lipasti, "Architectural support for server-side php processing," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 507–520, ACM, 2017.

[7] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 27–39, IEEE, 2016.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[9] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, *et al.*, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, 2011.

[10] G. Shi, M. Li, and M. Lipasti, "Accelerating search and recognition workloads with sse 4.2 string and text processing instructions," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 145–153, IEEE, 2011.

[11] M. S. B. Altaf and D. A. Wood, "Logca: a performance model for hardware accelerators," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 132–135, 2015.

[12] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 317–330, IEEE, 2019.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, p. 3, 2009.

[14] 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.

[15] D. Gope, *Architectural Support for Scripting Languages*. The University of Wisconsin-Madison, 2017.

[16] J. Choquette, O. Giroux, and D. Foley, "Volta: performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.