Abstract

Hardware Event Counters: Opportunities, Limits, and Bridging the Gap

Nicholas Alexander Lindsay

2026

The increasing number of hardware event counters available on commodity hardware should provide researchers fine-grained insight into microarchitectural behavior. Unfortunately, this opportunity is limited by unknown microarchitectural details and incomplete documentation. Consequently, researchers must make assumptions about the microarchitecture they profile, but these assumptions often fail to explain all the data - undermining derived conclusions.

This thesis introduces CounterPoint - a methodology and automated tool for finding self-consistent microarchitectural models that explain hardware event counter measurements. Models are expressed as flow charts, which are automatically converted to mathematical constraints evaluated against event counter measurements. Constraint violations act as refutations to microarchitectural assumptions, and facilitate a counter-example guided model refinement methodology. Statistical confidence regions mitigate measurement noise introduced by time-multiplexing events over a limited number of physical event counters. CounterPoint is fast, effective, and scales to dozens of event counters. We use CounterPoint to study the memory management unit on a commercial microprocessor, identifying undocumented/understudied features including page table walk merging, early paging structure cache lookup, and address translation prefetching.

CounterPoint can augment other research involving hardware event counters. Prior to CounterPoint's invention, we studied memory management unit performance across benchmarks with increasing memory footprints. The study shows that performance overhead often scales with the logarithm of the memory footprint, even with large pages. The overhead is distributed across the TLB, MMU cache, and hardware page table walker in a workload-dependent fashion. A significant fraction of initiated page table walks are either aborted or are on the wrong path of speculative execution. Using CounterPoint, we identified potential error sources that were unknown when the peer-reviewed study was published. This demonstrates that CounterPoint is an effective tool for evaluating microarchitectural research under evolving microarchitectural assumptions.

Hardware Event Counters: Opportunities, Limits, and Bridging the Gap

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Nicholas Alexander Lindsay

Dissertation Director: Abhishek Bhattacharjee

September 2026

To the giants upon whose shoulders I stand

Acknowledgments

TODO

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Opportunities and limits of hardware event counters

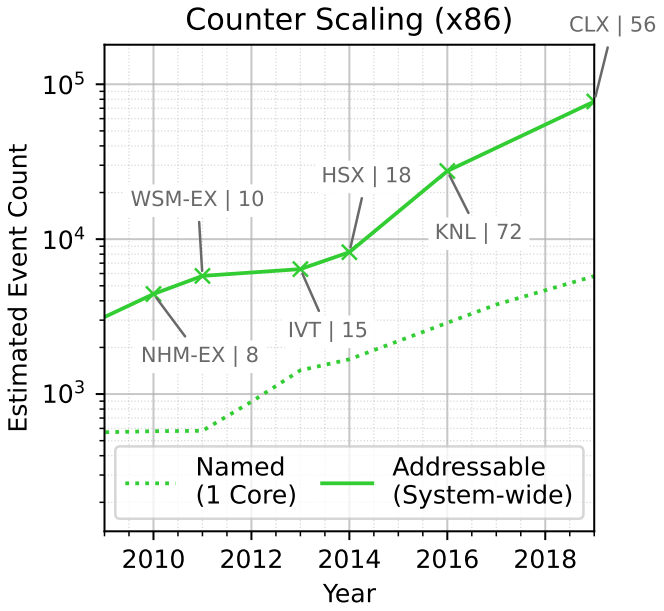


Figure 1.1: The estimated number of HECs events in x86-64 systems increased over $10\times$ between 2009 and 2019. (Y-axis plotted on a log-scale.) The dashed line shows the number of documented HEC 'names', assuming a single core. The solid line shows the number of 'addressable' events after accounting for per-core replication and the conservative removal of events that, while still documented (and potentially informative), have been deprecated by the vendor. Each data point represents a microarchitecture paired with its typical core count in a server system. This graph shows only documented events and does not include the thousands of additional undocumented HECs identified in recent work [152][1].

Hardware event counters (HECs) are specialized registers embedded in CPUs and hardware accelerators that provide low-overhead, fine-grained insights into microarchitectural behavior

during execution. First introduced in the 1980s—most notably in the DEC VAX and early RISC machines—HECs were originally designed to support performance tuning and system-level debugging.

Since then, their role has expanded. While HECs remain essential for identifying performance bottlenecks [153, 49, 85, 51], they are now also used to calibrate microarchitectural software simulators [76, 126], build analytical models of hardware [5, 10, 85], correlate microarchitectural activity with power and thermal behavior [69, 84, 128, 78, 33, 154], and more [2, 3, 114, 156, 11, 45, 144]. As their utility has grown, so has their prevalence: modern x86-64 processors now expose thousands of HECs —more than a 10× increase since 2009 (Figure 1.1).

The promise of a broad set of HECs. In principle, a rich set of HECs should allow experts to gain deeper insight into their mental model of the hardware, even without access to proprietary RTL or internal documentation. These insights are crucial for building accurate performance models and calibrating architectural simulators for future hardware [27, 24, 29, 25, 14, 18, 5, 124], and go beyond traditional HEC uses that measure only broad performance metrics like CPU and memory utilization [129, 91, 109, 133, 134, 52, 151].

Address translation provides a prime example. Modern processors devote many HECs to this function—for instance, IBM's Power9 includes 96 HECs dedicated solely to address translation [68]. Researchers often attempt to leverage these counters to reverse-engineer address translation hardware, enabling accurate integration into software simulators and analytical models of system performance [27, 24, 29, 25, 14, 18, 5, 124]. These models commonly assume specific behavior for the Paging Directory Entry (PDE) cache, which eliminates memory accesses to non-leaf page table levels during a walk. It is assumed that the PDE cache is accessed exactly once per page-table walk, implying that the number of PDE cache misses (`load.pde$_miss`) should not exceed page walks (`load.causes_walk`):

$$\texttt{load.pde\$\_miss} \leq \texttt{load.causes\_walk}$$

Surprisingly, our measurements on Intel Haswell show that this expected relationship—a sanity check of the expert's mental model, which we term a *model constraint*—does not always hold.

This challenges a widely held assumption in address translation research and casts doubt on the validity of much simulation-based work that relies on it. This example also illustrates two benefits of having a diverse set of HECs.

First, a diverse set of HECs helps detect violations of the model constraints associated with an expert's mental model of the hardware. Here, we can spot the violation only because Haswell exposes the `load.causes_walk` HEC —which many other processors lack. Without it, researchers rely on generic TLB miss HECs that miss such nuances.

Second, a diverse set of HECs helps explain *why* a model constraint is violated, enabling refinement toward a more accurate representation of the hardware. For example, comparing counters for retired TLB misses, PDE cache misses, and page table walks uncovers two likely undocumented behaviors: (i) merged walks to the same virtual address occurring after PDE cache lookup, and (ii) aborted translation requests that terminate after PDE cache lookup but before a page table walk begins. Without the full set of HECs, these effects would remain hidden.

The reality of a broad set of HECs. The PDE cache example is an ideal case where a diverse set of HECs helps reveal that there is a flaw in the expert's original mental model of the microarchitecture. In practice, however, experts are rarely able to leverage the full introspective power of HECs because they rely on manual and ad hoc approaches to doing so. In particular, they face two challenges:

First, understanding microarchitectural behavior requires identifying how HECs relate to one another—that is, determining *all* the model constraints that observed HEC data must satisfy to align with an expert's mental model (*i.e.*, their set of assumptions about the microarchitectural implementation). Our PDE cache model constraint illustrates how surprisingly difficult it can be to reason about even simple relationships involving just two HECs. As more HECs are used to check whether observed behavior matches expert expectations, the number of model constraints grows super-linearly (Figure 1.2). These model constraints become increasingly complex, often involving dozens of HECs in intricate relationships that are hard to reason about (Section 2.1).

[1]We counted the total number of HEC names in the Linux `perf` counter database [136, 66] to determine the set of 'Named' events per microarchitecture. We estimated the number of 'Addressable' events by (i) conservatively removing deprecated HECs, (ii) distinguishing between core and uncore HECs, and (iii) accounting for per-core replication. We account for per-core replication by summing together the uncore counters with the number of core counters multiplied by the typical core count of server systems of that microarchitecture.

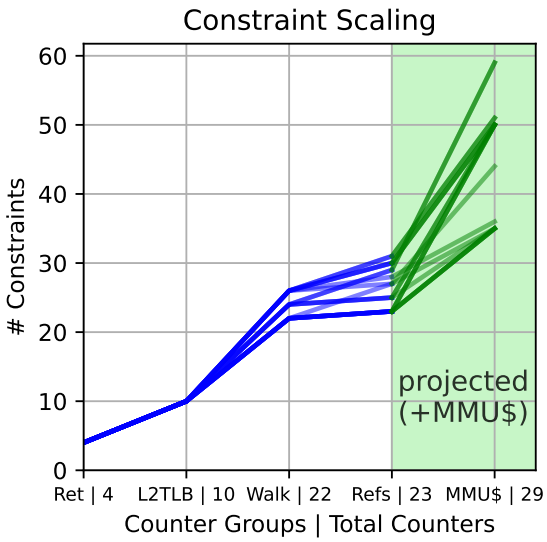


Figure 1.2: The number of model constraints implied by a model scales superlinearly with the number of HECs used to understand hardware behavior, and worsens significantly when including hypothetical HECs across all MMU caches (shown in green). Our x-axis shows increasing HEC count for an Intel Haswell MMU, in steps associated with all the HECs in a logical group; *e.g.*, 10 HECs for L2 TLB events.

Manual approaches to identifying and evaluating model constraints quickly become intractable. The challenge worsens when observed HEC values violate model constraints, forcing experts to revise their mental models—and then deduce entirely new, complex sets of associated model constraints.

Second, modern architectures allow recording thousands of *logical* HECs, but these are multiplexed onto a much smaller number of *physical* HECs —typically just 4 to 8 at a time. This means that HEC measurements are approximate rather than exact, leading to measurement noise that makes it even more difficult to evaluate model constraints. Multiplexing noise typically grows rapidly with the number of HECs being measured. Beyond a point, the growing number of HECs makes it nearly impossible to determine whether a representative model constraint is truly violated (Figure 1.3).

Extracting the promise of HECs with CounterPoint. To bridge the gap between the promise and reality of HECs, we invent CounterPoint[2]—a framework that helps experts reconcile HEC

[2]CounterPoint enables using hardware event *counters* to *point* out gaps in an expert's understanding of microarchitectures and makes it easier to explore improvements or *counterpoints* to their assumptions.

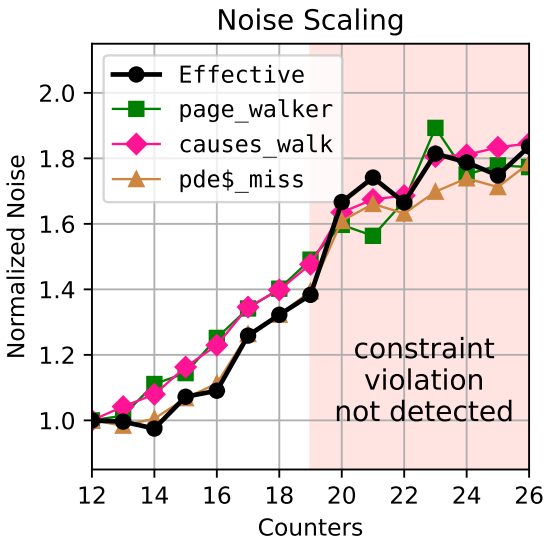


Figure 1.3: HEC measurement noise increases with active HECs; beyond a point, model constraint violations can no longer be reliably detected. For a representative model constraint on the Intel Haswell MMU ((1) in Table 2.1), we show that as measurement noise increases—both overall and for individual HECs —it becomes impossible to determine whether the model constraint is violated with 99% confidence once 19 HECs are active. Here, noise is defined as the standard deviation in the observed HEC values.

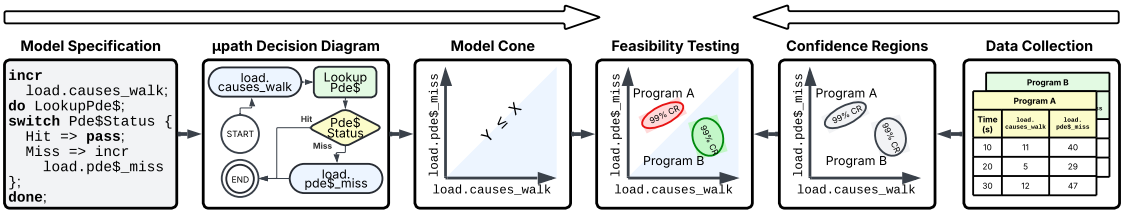


Figure 1.4: CounterPoint automatically determines the feasibility of a microarchitectural model against HEC data. Models are described using a DSL and transformed into a μpath Decision Diagram (μDD), which is analyzed to determine the model cone (the set of model constraints). Counter confidence regions are constructed for each observation to handle multiplexing noise. Observations are tested against all model constraints simultaneously. CounterPoint's counter confidence region bounds are sharper than other approaches, enabling more violations to be identified, and thereby enabling more opportunities to refine the expert's microarchitectural assumptions. CounterPoint effortlessly supports dozens of HECs and constraints.

data with their mental models of the microarchitecture. CounterPoint automates the demanding task of generating all model constraints associated with a model and checking them against noisy HEC data, enabling exploration of the accuracy of a wider range of microarchitectural models. CounterPoint is centered around three key insights

First, experts can more naturally express their envisioned hardware as a directed acyclic graph (DAG) linking hardware components to HEC activity, rather than directly constructing model constraints. DAGs are well-suited to formal tools that can automatically derive these constraints. We introduce the μpath Decision Diagram (μDD)—a specialized DAG for capturing an expert's mental model of microarchitectural structures and how interactions among these structures increment HECs. A μDD concisely describes a set of microarchitectural execution paths (μpaths) that micro-ops (μops) may follow. Each μpath is associated with specific microarchitectural events, including those that increment performance counters, enabling natural and complete generation of all constraints implied by the model.

Second, experts can refine microarchitectural models more effectively when model constraints are as tightly upper- and lower-bounded as possible (*e.g.*, constraint (3) in Table 2.1 is most useful when its left-hand side tightly lower-bounds the number of memory references in a page table walk). Tight constraints increase sensitivity to even minor deviations from the expert's mental model. Such tightness is more likely when HEC relationships are expressed at the granularity of micro-ops, enabling precise attribution of events to specific hardware behaviors. μDDs naturally exploit this by modeling micro-op flows through execution paths, yielding model constraints with inherently tight bounds.

Third, while more HECs increase multiplexing noise, they also increase intrinsic correlations (*e.g.*, page table walks often correlate with TLB misses). These correlations allow statistical methods to build tight *counter confidence regions*—ranges of HEC values likely to occur with a given probability from noisy data. Compared to traditional methods that treat counter noise independently [91, 10], this approach substantially reduces the impact of noise, enabling CounterPoint's automated analysis to scale well beyond the number of physically available HECs.

The CounterPoint approach. Experts begin by expressing their mental model of the microarchitecture in a domain-specific language, which CounterPoint translates into a μDD (Figure 1.4).

Experts also run workloads on the target hardware, collecting as many active HECs as needed for analysis.

Given a μDD, CounterPoint applies convex geometry techniques to derive the *model cone*—all HEC value combinations producible by micro-ops traversing the μDD. The model cone represents the values that simultaneously satisfy all model constraints, and eliminates the need for manual derivation. CounterPoint then processes noisy HEC measurements from real hardware, extracting intrinsic correlations to define tight *counter confidence regions*: ranges of HEC values inferred with high confidence despite multiplexing noise.

Finally, with feasibility testing, CounterPoint compares the counter confidence region against the model cone. If they do not intersect, the expert's model is inconsistent with the HEC observations, implying that some model constraints are violated. CounterPoint reports these violations, guiding how the μDD may be revised for consistency. This enables iterative exploration: the expert proposes new μDDs, and CounterPoint tests them until a consistent model is found.

Evaluating CounterPoint via a case study. We demonstrate CounterPoint's capabilities by applying it to the Intel Haswell Memory Management Unit (MMU), where we uncover several previously undocumented and underdocumented features. These include a load–store queue-side TLB prefetcher (as well as its trigger conditions and interaction with page hotness tracking), hardware mechanisms that merge and abort page table walks, and a cache for the root level of the page table. The Haswell MMU serves as a compelling case study: it embodies complex hardware–software interactions, and has been foundational for a decade of address translation research [113, 99, 9, 150, 140, 50, 5, 157, 135, 94, 85]. Yet, it is poorly modeled in state-of-the-art software simulators [148, 87, 48, 47], motivating recent efforts to use HECs to reverse-engineer accurate models [27, 24, 29, 25, 14, 18, 5, 124]. As a rigorous test of CounterPoint's analysis of sophisticated microarchitectural behavior, the Haswell MMU case study provides a foundation for extension to other components and more modern microarchitectures.

Situating CounterPoint within the microarchitectural research space. CounterPoint can influence the wider space of microarchitectural research performed with hardware event counters. To illustrate this, we first present a peer-reviewed research study [85] we published prior to CounterPoint's existence. Using hardware event counters, we profiled how Haswell MMU

performance scales with increasing memory footprint across a number of translation-intensive workloads. We found that: address translation overhead often scales logarithmically with memory footprint (even when large pages are used), that this overhead is distributed across MMU components, and that a significant number of page table walks are aborted or misspeculated.

We revisit these findings in light of microarchitectural features we later identified with CounterPoint. We find that the high level conclusions likely hold, but that the microarchitectural behaviors identified with CounterPoint may introduce subtle errors in how overhead is attributed to individual MMU components. Overall, this demonstrates that CounterPoint can help evaluate the strengths and weaknesses of research studies that use hardware event counters.

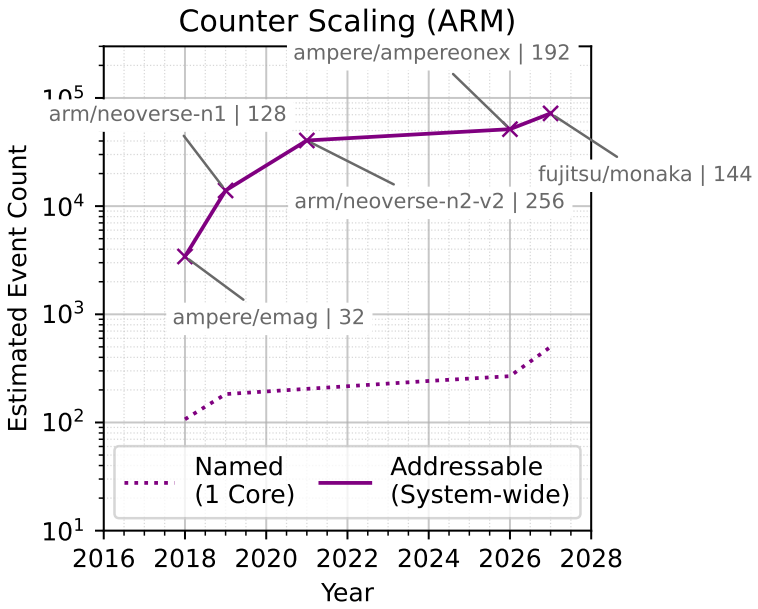
1.2 Background and related work

1.2.1 Hardware event counters

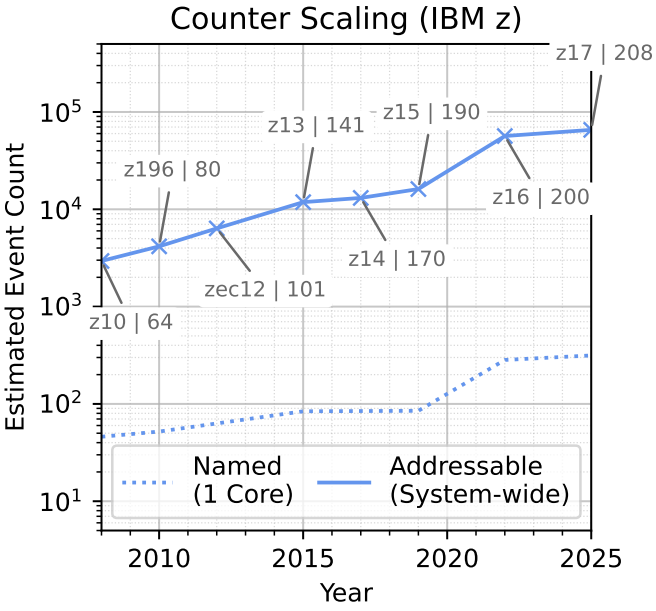
Trends in event counts. The number of events recordable by HECs has grown dramatically since 2008 across multiple CPU architectures including x86 (Figure 1.1), ARM (Figure 1.5a), and IBM (Figure 1.5b). This growth is driven by an increase in the number of named events, alongside a general increase in core counts. The presented event counts - which we derive from the Linux `perf` database [136] - are likely understimates because they do not include undocumented events [152]. The current abundance of recordable events presents a golden opportunity for researchers to characterize microarchitectural behaviors and their evolution across microprocessor generations.

Reverse-engineer applications. Computer architects have used HECs to reverse-engineer specific microarchitectural features. HECs are used by uops.info [2] and nanoBench [3] to reverse-engineer micro-op performance and port assignment, as well as cache replacement policies. Several studies have focused on reverse engineering the MMU [12, 143, 50, 157], while Ragab et al [122] use HECs to characterize the security implications of machine clears. Binoculars [157] use HECs to characterize page table walker contention. With μDD models, this thesis presents a more general-purpose approach.

Multiplexing noise. Multiplexing noise is a well-studied problem [91, 10, 13]. Azimi et al [11] quantify multiplexing noise for a range of workloads. CounterMiner [91] replacing outliers with



(a) Hardware event counter scaling on Arm processors between 2018 and 2027[3].



(b) Hardware event counter scaling on IBM Z processors between 2008 and 2025[4].

Figure 1.5: Hardware event counter scaling on Arm (purple) and IBM Z (blue) processors between 2008 and 2027. Dashed lines indicate documented HEC 'names' and solid lines indicate the number of 'addressable' events after accounting for per-core replication and the conservative removal of events that, while still documented (and potentially informative), have been deprecated by the vendor. Each data point represents a microarchitecture paired with a representative core count for that microarchitecture. This graph shows only events officialy documented in the Linux `perf` database [136].

interpolated values. BayesPerf [13] reduces noise by exploiting *known* statistical relationships between counter values. This thesis infers correlations to reduce noise impact.

Interpreting HEC values. Interpreting HEC values correctly remains challenging. Vendors provide explicit metrics that convert HEC values to standard metrics (*e.g.*, CPI, MPKI, hit rates, *etc.*), but not all HECs are used. The Counter Inspection Toolkit [43] and related work [16, 17] correlate counters with individual microbenchmarks to define new metrics. Top Down Methodology [153] employs metrics and thresholds to enable application developers to identify performance bottlenecks. Unlike bespoke approaches, CounterPoint μDDs capture both HEC semantics and microarchitectural features by construction.

1.2.2 Formal methods for modelling microarchitectures

Formal modeling for microarchitectures has recently been used for memory consistency [92, 89, 90, 63, 104, 64], cache coherence [118, 119, 106, 107, 108, 44], and security [137, 65]. CheckSuite and related tools [92, 89, 90, 63, 104, 64, 137, 65] describe microarchitectural executions using μspec models featuring μpaths and inter- and intra-μpath dependencies. The formalism presented in this thesis is compatible with these approaches.

1.2.3 Address translation

Address translation is the process of converting application-visible *virtual addresses* to hardware *physical addresses* [29, 61]. This process involves navigating an in-memory data structure called the *page table* during a *page table walk*. On x86-64 this structure is implemented as a 4- or 5-level radix tree and is accessed in hardware by a *page table walker*. Each node in the tree is called a *page table entry*, or PTE for short. The address translation process must be performed on every application memory access.

Since page table walks are time-consuming, processors include specialized caches to avoid them. *Translation lookaside buffers* (TLBs) cache the results of the most recent page table walks,

[3]For ARM, the number of 'Addressable' events was calculated by multiplying the number of non-deprecated events by the representative core count, followed by adding any documented cluster- or system-wide counters in the `perf` database for the given microarchitecture.

[4]For IBM z, the number of 'Addressable' events was calculated by multiplying the number of non-deprecated events by the representative core count.

thus storing directly the mapping from virtual to physical addresses. Application accesses can often be translated by the TLB alone, completely eliminating a page table walk and hence injecting no additional memory accesses into the program.

To improve performance on TLB misses, modern memory management units (MMUs) incorporate *MMU caches*. On Intel x86-64 processors, these structures are called *paging structure caches* [67]. Paging structure caches cache partial page table walks [15], allowing page table walks to skip accesses at or near the top of the radix tree.

Traditionally, address translation is performed on the granularity of 4KB *pages*. Contiguous blocks of aligned 4KB application visible virtual addresses map to contiguous blocks of aligned 4KB hardware visible physical addresses. However, modern architectures also contain larger page sizes called *superpages*. On x86-64 these are 2MB and 1GB in size. Superpages have two benefits. First, they map a greater region of an application's virtual address space in a single PTE, increasing the effective capacity of the TLB and hence reducing the TLB miss rate. Second, they reduce the length of page table walks since PTEs can be found higher up the radix tree.

For a comprehensive treatment of address translation hardware on modern microprocessors, see Bhattacharjee and Lustig [29].

1.3 Contributions of this thesis

This thesis advances HEC-driven microarchitectural research, and the academic community's understanding of address translation hardware, by:

- Defining HEC model constraints and demonstrates their ability to expose hidden microarchitectural behavior.
- Introducing the μDD, a compact representation that encodes both microarchitectural assumptions and HEC semantics.
- Defining the model cone, showing how it can be naturally derived from the μDD, and proving its equivalence to the model constraints of the μDD.
- Applying counter confidence regions to mitigate measurement noise, enabling reliable inference even when the number of counters exceeds hardware limits.

- Developing μDD feasibility testing to automatically validate measured HEC data against a μDD's implied constraints.

- Revealing several likely undocumented and underdocumented features in the Intel Haswell microarchitecture — including TLB prefetching, early paging-structure cache lookups, and merged page table walks — using CounterPoint's automated analysis.

- Quantifying the performance of Haswell MMU components under workloads with increasing memory footprint using a hardware event counter based analysis.

- Demonstrating how CounterPoint can help evaluate the strengths and weaknesses of microarchitectural research that uses hardware event counters.

Chapter 2

CounterPoint

This chapter is adapted from "CounterPoint: Using Hardware Event Counters to Refute and Refine Microarchitectural Assumptions" which we published at ASPLOS'26. This paper was joint work with Abhishek Bhattacharjee, Caroline Trippel, and Anurag Khandelwal.

2.1 Models and constraints

Microarchitectural assumptions and model constraints. We refer to a set of microarchitectural assumptions as a *microarchitectural model.* A *model constraint* is an equality or inequality defined over HEC values that must be satisfied for the HEC values to be consistent with the microarchitectural model. We call them "model constraints" because they *constrain* the set of HEC values that agree with the microarchitectural assumptions.

The pros and cons of model constraints. Model constraints are valuable because they let experts identify exactly when and how their assumptions about the microarchitecture break down. The HECs involved in a violated model constraint highlight which parts of the model may be incorrect. However, to be fully effective, all (often dozens of) model constraints must be enumerated, and each must be correct and tight. By tight, we mean the bounds leave minimal slack: loose constraints can miss infeasible observations, whereas tight constraints clearly delineate what is possible versus impossible, making violations easier to detect.

Manually deriving all the constraints is onerous, even for an expert. Table 2.1 shows just a subset of constraints for a simple Intel Haswell MMU model. Each may involve many HECs and depend on the intersection of multiple microarchitectural assumptions.

Table 2.1: The Haswell MMU requires reasoning about dozens of model constraints; we show three representative examples. Deriving the exact constraints for a given model is challenging because they stem from subtle microarchitectural assumptions. For example, Constraint 1 relies on expert knowledge that no retired TLB miss suffered a prior page fault. Constraint 2 relies on even more subtle knowledge that an upper bound on the number of memory references injected by a page table walker is determined by (i) PDE cache hit/miss status; (ii) the page size of the translation and (iii) the fact that every walk makes at least one memory reference. For brevity, we define: $\texttt{walk\_ref} \triangleq \texttt{walk\_ref.l1} + \texttt{walk\_ref.l2} + \texttt{walk\_ref.l3} + \texttt{walk\_ref.mem}$.

(1) $\texttt{load.ret\_stlb\_miss} \leq \texttt{load.walk\_done}$	**2 HECs**
Every TLB-miss micro-op that retires must have obtained a valid translation from a page-table walk.	
(2) $\texttt{walk\_ref} \leq \texttt{load.causes\_walk} + \texttt{store.causes\_walk} + 3 \cdot \texttt{load.pde\$\_miss} + 3 \cdot \texttt{store.pde\$\_miss} - \texttt{load.walk\_done\_2m} - \texttt{store.walk\_done\_2m} - 2 \cdot \texttt{load.walk\_done\_1g} - 2 \cdot \texttt{store.walk\_done\_1g}$	**12 HECs**
The number of memory accesses made by the page table walker is upper bounded by the distribution of combinations of page sizes and PDE cache interactions.	
(3) $\texttt{load.causes\_walk} + \texttt{store.causes\_walk} + \texttt{load.walk\_done\_1g} + \texttt{store.walk\_done\_1g} \leq \texttt{walk\_ref}$	**8 HECs**
Every page table walk must result in one or more page table walker memory accesses. Walks that complete with 1GB page emit two memory references when the MMU cache for the root page table level is absent.	

Worse, constraints are easy to formulate either too loosely or incorrectly. For example, one might bound the number of page walker loads on Haswell, with its four-level page table:

$$\texttt{walk\_ref} \leq 4 \cdot (\texttt{load.causes\_walk} + \texttt{store.causes\_walk})$$

This is correct but not tight, since it ignores page size and MMU cache hits (unlike Constraint 2 in Table 2.1).

Alternatively, one could try to exploit the fact that larger pages shorten page table walks:

$$\begin{aligned}\texttt{walk\_ref} \leq\ & \\ & 4 \cdot \texttt{load.walk\_done\_4k} + 4 \cdot \texttt{store.walk\_done\_4k} + \\ & 3 \cdot \texttt{load.walk\_done\_2m} + 3 \cdot \texttt{store.walk\_done\_2m} + \\ & 2 \cdot \texttt{load.walk\_done\_1g} + 2 \cdot \texttt{store.walk\_done\_1g}\end{aligned}$$

But this version is too strong: it rejects valid cases where walks inject memory accesses but do not

terminate (*e.g.*, invalid translations). As Constraint 2 shows, the tightest correct bound is actually a far more nuanced relationship.

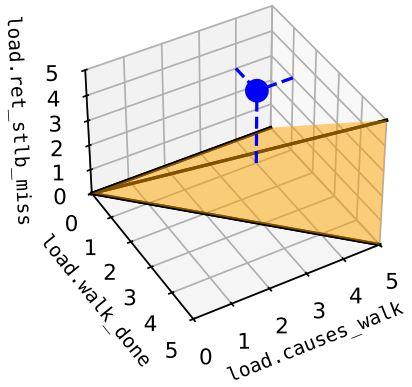
Even simpler constraints require tightness. For instance, Figure 2.1a shows an infeasible observation of HEC values detectable only with enough relevant constraints. With fewer or irrelevant counters (Figures 2.1b and 2.1c), the violation slips through. When scaling to dozens of model constraints, many of which include complex relationships among dozens of HECs each, all these problems compound.

The geometry underlying model constraints. Model constraints are powerful for validating hardware assumptions but are unscalable as they are derived in an ad hoc manner. The challenge is not in their use, but their derivation.

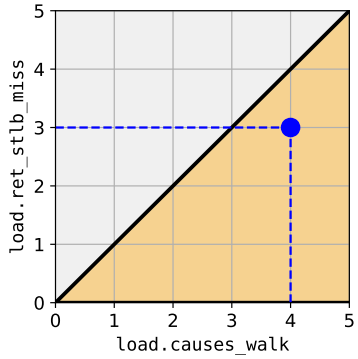
We observe that model constraints naturally arise because microarchitectural events occur in predefined groups rather than in isolation—for example, each completed walk for a 4KB page (`load.walk_done_4k`) involves 1 to 4 page table walker memory accesses (`walk_ref`). Our insight is that instead of manually deriving constraints, experts can more easily enumerate all valid groupings, letting CounterPoint automatically determine whether an observed set of events could result from some combination of groups. This approach enables scalable feasibility checking of the model constraints.

We enable experts to specify how μops interact with the microarchitecture—including their effect on HECs —using μDDs. A μDD is a specialized DAG where each path represents a single HEC group, enabling automated testing of observed HEC values against feasibility constraints. μDDs are centered on μops because they form a natural unit for grouping microarchitectural events: they are familiar to experts, fine-grained enough to capture low-level hardware interactions, and directly responsible for incrementing HECs. The DAG representation is concise; a few nodes can efficiently describe an exponential number of μpaths.

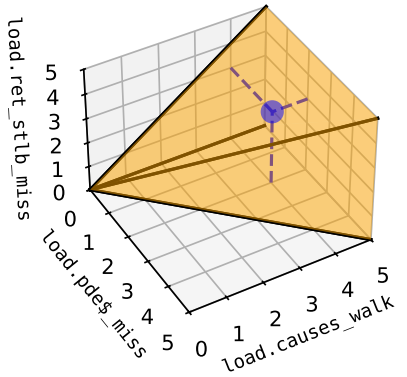
The group-matching problem naturally induced by the μDD is fundamentally a counting problem that can be framed in terms of convex geometry. The resulting geometric object—*i.e.*, the model cone—represents all valid combinations of HEC values. The Minkowski–Weyl theorem from computational geometry states that every model cone has two equivalent representations: one as the set of points generated by a μDD, and the other as the set of points bounded by model



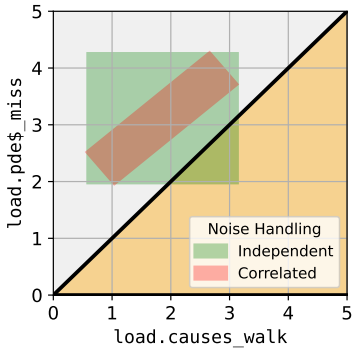
(a) Candidate model cone (yellow) from three HECs shows a violation of a model constraint.



(b) Ignoring an HEC loosens the model cone and misses the violated constraint in Figure 2.1a.



(c) HECs with subtly different semantics loosen the cone, missing Figure 2.1a's violated constraint.



(d) Counter confidence regions are tighter with HEC correlations (red) than without (green).

Figure 2.1: The ability of HECs to test assumptions depends on their number and semantics, shown here pictorially. The orange regions represent points which satisfy all model constraints; the blue dot represents an observation; the red and green boxes represent two alternative constructions of counter confidence regions. Model constraints correspond to edges in 2D or faces in 3D. (a) Consider a model cone constructed from the three HECs shown. These counters imply three constraints: `load.ret_stlb_miss` $\leq$ `load.walk_done` because each retired STLB miss must correspond to a unique, successfully completed page table walk; `load.ret_stlb_miss` $\leq$ `load.causes_walk` because each retired STLB miss must trigger exactly one page table walk; and `load.walk_done` $\leq$ `load.causes_walk` because only a subset of initiated page table walks ultimately complete. The first two inequalities rely on the assumption that STLB misses are never merged. Using all three HECs clearly exposes a violation of these constraints, indicating a flaw in the expert's mental model. (b) All three HECs were required to detect this flaw; removing `load.walk_done` eliminates the second and third constraints, making the model violation undetectable. (c) Simply substituting `load.walk_done` with `load.pde$_miss` (or any other counter) is insufficient, because the semantics of each counter matter. Using this alternative counter adds the constraint `load.pde$_miss` $\leq$ `load.causes_walk`, but this constraint still fails to reveal the model violation. (d) Counter confidence regions replace point observations with value ranges; exploiting correlations yields tighter bounds than assuming independence.

constraints [53]. We leverage these dual representations by allowing the expert to express their microarchitectural assumptions in the form most natural to them—by encoding their hardware assumptions in a μDD—while enabling CounterPoint to automatically deduce model constraints as required for user feedback. CounterPoint derives the model constraints using a custom algorithm which calls into an off-the-shelf *convex hull* solver, as described in Section 2.5.

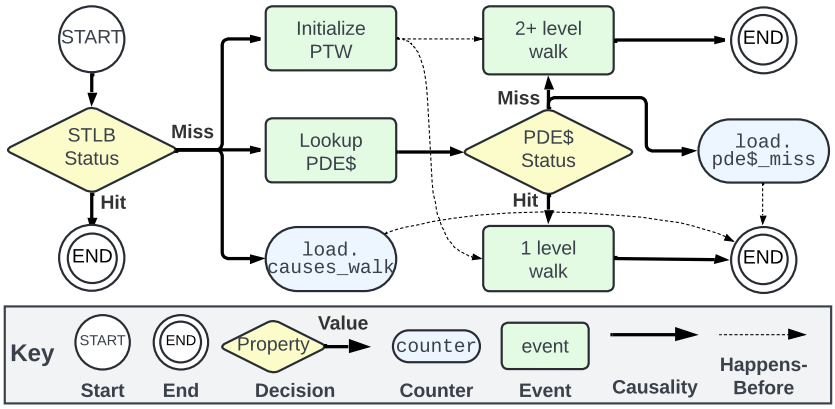
Generating tight confidence regions of HEC observations. Identifying flaws in the expert's mental model requires not only a tight model cone but also computing the narrowest possible range of values that can be confidently inferred from the observed HEC values, despite multiplexing noise.

Standard measurement tools (*e.g.*, `perf`) report the mean and standard deviation of the samples for each HEC, which can be used to construct counter confidence regions. Naive methods assume each HEC is independent, resulting in overly loose counter confidence regions (Figure 2.1d, green box) that reduce the ability to detect violations of model constraints.

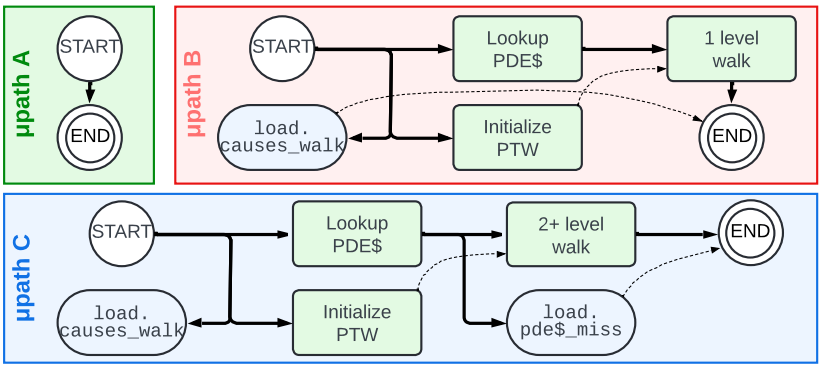
Instead, we discover that HEC values are often correlated, a finding we extract from time-series measurements. These correlations mean that the data typically have far fewer degrees of freedom than the number of counters, allowing us to construct much tighter counter confidence regions—even when dozens of counters are measured (Figure 2.1d, red box). Tighter counter confidence regions uncover more accurate microarchitectural models.

Feasibility testing for guided model exploration. High-quality models demand detail, with μDDs describing hundreds of unique execution paths. This produces tight model constraints, but also drives rapid growth in complexity. CounterPoint uses linear programming to efficiently determine model feasibility, and a conic hull algorithm (Section 2.5) to derive model constraints when infeasible observations occur. Violated model constraints are reported to the expert, who uses this information to formulate refined μDDs that resolve these discrepancies and represent more accurate models of the hardware.

Naturally, the precision which CounterPoint can infer details about the microarchitectural features depends on having a dataset of HEC observations from a rich and diverse set of programs that stress all relevant corners of the microarchitecture. Consequently, to uncover details of the Haswell MMU in our case study, we evaluated about 20 million HEC samples across a diverse



(a) Example μDD that describes how a μop interacts with the STLB and PDE cache, in the presence of `load.causes_walk` and `load.pde$_miss` HECs. This μDD encodes three μpaths.



(b) This μDD describes three unique μpaths, each corresponding to different assignments to microarchitectural properties (*e.g.*, *STLB Status* and *PDE$ Status*). Edges represent happens-before order.

Figure 2.2: A μDD encodes a set of microarchitectural execution paths (μpaths). Each μpath describes a set of events per μop.

range of workloads with dozens of models, each exhibiting a unique combination of bespoke microarchitectural features. As we use CounterPoint to refine our understanding of the hardware, we continue to expand this set of models.

2.2 From diagrams to model cones

μpath decision diagrams. A μDD encodes a set of microarchitectural execution paths that individual μops may take through part of the microarchitecture (Figure 2.2a is an example encompassing a subset of address translation hardware). At the core of CounterPoint is the concept of a microarchitectural execution path, or μpath: a happens-before ordered set of hardware events

induced by a μop [65].

Micro-paths are derived from a μDD by performing a graph search along CAUSALITY edges, with nodes and CAUSALITY edges added to the μpath as they are encountered. When a DECISION node is encountered, there are two possibilities. If the property has been assigned a value (determined by the labels on outgoing edges) earlier in the traversal, then the corresponding outgoing CAUSALITY edge is followed. Otherwise, a concrete property value from the outgoing CAUSALITY edge labels is selected and the corresponding edge is followed. This process continues until all nodes and CAUSALITY edges have been added. HAPPENS-BEFORE edges between node pairs are instantiated in the μpath if there exists a HAPPENS-BEFORE edge between the corresponding μDD nodes.

When exhibiting a μpath during its execution, a μop generates events in a time order that respects both CAUSALITY and HAPPENS-BEFORE edges. Events come in two forms: standard EVENT nodes (green boxes), which represent standard microarchitectural events, and COUNTER nodes (blue pills), which correspond to events directly recorded by HECs.

μpath counter signatures. Each μpath has an associated counter signature—a vector that records how many times each HEC appears within a μpath. This signature captures how a μop following that μpath increments the HECs.

Counter flow equation. Our first goal is to precisely define when an observed set of HEC values is "feasible" with respect to a μDD. We do so via the *counter flow equation*, which links HEC values to the number of μops traversing microarchitectural execution paths.

The key insight behind the counter flow equation is that μops increment HECs as they traverse a μpath, creating a direct relationship between the microarchitectural *flow* of μops and the resulting HEC values.

Let $\mathcal{D}$ be a micro flow diagram and $\mathcal{P}(D)$ the set of μpaths it encodes. A microarchitectural *flow* $f(\cdot)$ assigns each μpath $p \in \mathcal{P}(D)$ a non-negative number of μops traversing it. Each μop on μpath p increments the HECs according to the μpath 's *counter signature* $\vec{S}(p)$—the vector of counter occurrences along p. Thus, the contribution of μpath p is $f(p) \cdot \vec{S}(p)$, and the total HEC

value vector $\vec{v}$ is the sum over all μpaths:

$$\vec{v} = \sum_{p \in \mathcal{P}(D)} \vec{S}(p) \cdot f(p) \qquad \text{(Counter Flow Equation)}$$

This *counter flow equation* links observed HEC values to the flow of μops through the μDD, and is only valid when $f(p) \geq 0$ for all μpaths, as negative flows of μops are impossible. Intuitively, the final counter values are given by the total number of HEC increments across all dynamic μop instances.

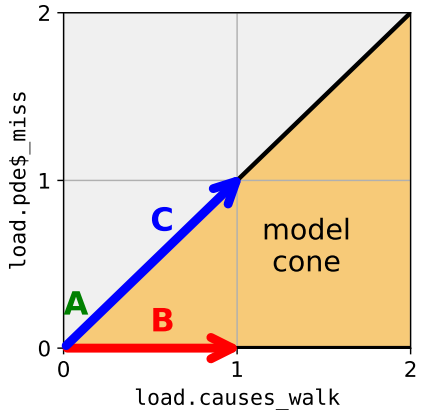


Figure 2.3: The model cone is determined purely by μpath counter signatures.

Deriving the model cone. The model cone is the set of all HEC value combinations generated by valid microarchitectural executions (*i.e.*, those with *non-negative flows*). We determine observation feasibility by testing if it lies within the model cone, a task accomplished using linear programming. This means that feasibility can be determined even without knowing $f(\cdot)$ exactly.

Mathematically, we define a model cone $K_{\mathcal{D}}$ for a μDD $\mathcal{D}$ as the set of HEC values that are generated by microarchitectural executions with non-negative flow:

$$K_{\mathcal{D}} \triangleq \left\{ \sum_{p \in \mathcal{P}(D)} \vec{S}(p) \cdot f(p) \;\middle|\; f(p) \geq 0 \right\} \qquad \text{(Model Cone)}$$

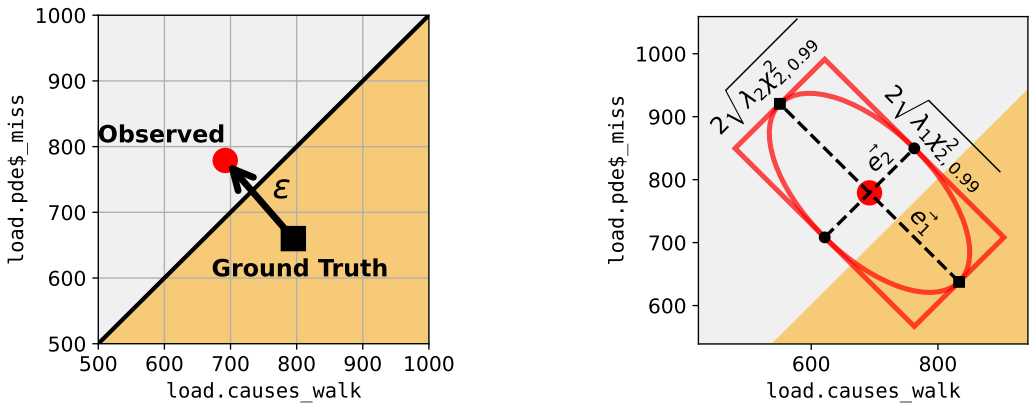
Geometrically, $K_{\mathcal{D}}$ is a convex[1] polyhedral[2] cone[3] defined purely by the μpath counter signatures

[1] **Convex**: If $x, y \in K_{\mathcal{D}}$ then $\alpha x + (1-\alpha)y \in K_{\mathcal{D}}$ for $0 \leq \alpha \leq 1$.
[2] **Polyhedral**: Defined by a *finite* number of equalities and inequalities.
[3] **Cone**: If $x \in K_{\mathcal{D}}$ then $\alpha x \in K_{\mathcal{D}}$ for $\alpha > 0$.

in the μDD (Figure 2.3). Intuitively, the model cone represents the space of all allowed HEC combinations.

Generalizability. Our decision to design CounterPoint so that it links fine-grained microarchitectural events and interactions—represented as μpaths of μops—with HEC updates is intentional. μop-centered execution paths have been used extensively in prior work to model low-level hardware, including formal verification of memory consistency and its interactions with coherence and virtual memory [88, 90, 93, 64], side-channel security [137, 65], and functional correctness [88, 65]. Because these approaches cover many aspects of CPU pipelines, they suggest that CounterPoint is well positioned to extend to other microarchitectural components.

2.3 Feasibility testing with noise



(a) Noise introduced by multiplexing can make valid combinations appear infeasible.

(b) Confidence region construction oriented along eigenvectors of covariance matrix.

Figure 2.4: Testing observations for inclusion in the model cone is complicated by noise, which can cause observations to spuriously appear infeasible (Figure a). CounterPoint handles noise by constructing confidence regions at the 99% confidence level (Figure b). The counter confidence region is an ellipsoid which CounterPoint approximates by its bounding box, enabling a linear programming formulation. The scale and orientation of the confidence region is determined by (i) the confidence level and (ii) correlations in the observed data. λ_k and $\vec{e_k}$ denote the kth eigenvalue/eigenvector of the estimated covariance matrix.

An observation is feasible if it resides within the model cone; a problem solvable with linear programming. Unfortunately, observations are subject to multiplexing noise which must be accounted for to prevent false violations (*e.g.*, Figure 2.4a).

Handling noise with counter confidence regions. Counter confidence regions handle noise by treating each observation not as a single value, but instead as a point drawn from a set of values within which the true value is likely to occur, given the presence of noise in the measurement. The likelihood of the region capturing the true value is given by the confidence level, which we fix to 99% for our analyses. The size and shape of the counter confidence region depends on parameters of the underlying distribution, which can be inferred from the HEC measurements themselves. CounterPoint computes covariances (in addition to means and variances computable by perf), producing tight counter confidence regions that are more likely to catch violated constraints.

CounterPoint requires HEC vector samples $\{Y_i\}_{i=1}^{M}$ recorded at regular time intervals (*e.g.*, every 10 seconds) over the course of a program's execution. Such functionality is provided by standard tools (*e.g.*, `perf`). CounterPoint computes the sample mean $\bar{Y}$ as a HEC vector representative of the entire execution. Statistically, the sample mean is drawn from a Gaussian distribution per the Central Limit Theorem. With Gaussian distributions, the confidence region is fully determined by the sample mean and sample mean covariance [60]. We calculate the HEC covariance matrix Σ_Y. We estimate the sample mean covariance with the plugin estimator $\Sigma_{\bar{Y}} = \frac{1}{M}\Sigma_Y$. This defines the confidence region:

$$\left\{ \vec{v} \;\middle|\; (\vec{v} - \bar{Y})^T \Sigma_{\bar{Y}} (\vec{v} - \bar{Y}) \leq \chi^2_{N,\alpha} \right\} \qquad \text{(Confidence Ellipsoid)}$$

Intuitively, this means that the confidence region is an ellipsoid in shape, and that the ground truth (*e.g.*, noise-free) counter value is contained within the ellipsoid with $(1 - \alpha)$-confidence. We adapt it to a linear program in the following section. The confidence region can be made tighter by obtaining more samples (*e.g.*, with longer running programs), providing the program has consistent steady-state behavior.

Determining feasibility with a linear program. Given a model cone and a counter confidence region, we can assess the feasibility of an HEC observation at a specified confidence level. If the counter confidence region intersects the model cone, the observation is deemed feasible. If there is no intersection, the observed HEC values must violate at least one model constraint at that confidence level. For example, Figure 2.4b shows a counter confidence region which intersects

with the model cone, indicating a feasible observation.

To test for feasibility, CounterPoint uses linear programming because of its efficiency, relative simplicity, and availability in mature software libraries [101, 46, 6, 37]. CounterPoint constructs a linear program[4] by instantiating non-negative variables for the flow and counter values (see Section 2.2). The flows and counter values are related by the Counter Flow Equation, implicitly describing the model cone. The counter confidence region, being a quadratic form, cannot be directly encoded. Instead, we approximate it with a bounding hyper-rectangle (Figure 2.4b). This bounding box is aligned with the principal components of the data, producing the tightest rectangular bound on the confidence region. Empirically, our bounding box approximation detects many surprising constraint violations (Section 2.6). We leave alternatives like quadratic programming for future work.

2.4 Guided model exploration

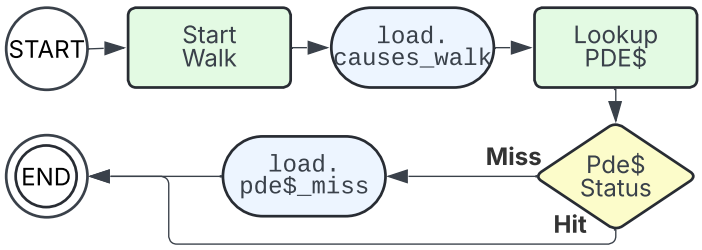
Discovering microarchitectural features. The specific model constraints that are violated guide the expert in identifying which microarchitectural features need to be added or modified in the μDD to make it feasible. When a model constraint of the form $a \cdot x \leq b \cdot x$ is violated, then for all feasible μDDs, there must exist a μpath whose μpath counter signature $\vec{S}(p)$ satisfies $a \cdot \vec{S}(p) > b \cdot \vec{S}(p)$.

We illustrate this with an example (Figure 2.5). Figure 2.5a is a simple μDD for load μops upon a TLB miss. We assume that the load μop first initializes the page table walker - incrementing `load.causes_walk-` before looking up the PDE cache. In the event of a cache miss, `load.pde$_miss` is incremented. This model implies model constraint $\mathcal{C}$ (2.5b).

CounterPoint identifies that Constraint $\mathcal{C}$ is violated by one or more HEC observations[5]. Therefore there are workloads where `load.pde$_miss` exceeds `load.causes_walk`. To explain this apparent contradiction, we must introduce one or more microarchitectural features into the μDD that allow for this constraint to be broken. This corresponds to modifying the μDD such

[4]The linear program is provided in Appendix A.

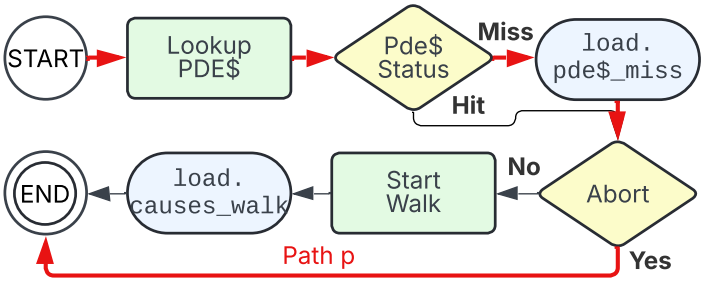
[5]When an observation is deemed infeasible with respect to an μDD, CounterPoint automatically tests the observation against each feasibility constraint to identify violations. Deriving and testing the feasibility constraints is a non-trival procedure; we describe our implementation in Section 2.5.



(a) Initial model.

$$\mathcal{C} \triangleq \texttt{load.pde\$\_miss} \leq \texttt{load.causes\_walk}$$

(b) Violated model constraint.



(c) Refined model.

Pde$ Status	Abort	load. causes_walk	load. pde$_miss	Satisfies $\mathcal{C}$
Miss	Yes	0	1	No

(d) Properties of μpath p including μpath counter signature.

Figure 2.5: Modifying μDD to remove constraint violations is equivalent to identifying candidate microarchitectural features. (a) Initial μDD of page table walk. (b) Model implies this model constraint, which is violated. (c) μDD is updated by (i) assuming PDE cache is looked up prior to starting walk, and (ii) allowing translation requests to be aborted before starting a walk. (d) Model no longer implies constraint $\mathcal{C}$ as $\vec{S}(p)$ does not satisfy constraint.

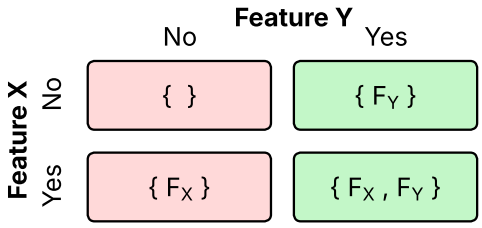


Figure 2.6: Microarchitectural models (boxes) classified by their features and consistency with hardware performance counter data. Red: inconsistent. Green: consistent.

that it contains μpath(s) whose μpath counter signatures explicitly violate $\mathcal{C}$.

One way to resolve this is to assume that (i) the PDE cache is accessed before starting a page table walk, and (ii) translation requests can be aborted between the PDE cache lookup and the start of the walk. This allows lookups to access the PDE cache without incrementing `load.causes_walk`. Applying these assumptions produces a new μDD (Figure 2.5c), with a new μpath p whose μpath counter signature $\vec{S}(p)$ explicitly violates constraint $\mathcal{C}$ (Figure 2.5d). Analysis of this μDD confirms that $\mathcal{C}$ is no longer implied, resolving the violation. In practice, many constraints often need resolution, requiring an iterative μDD refinement process.

Classifying microarchitectural models. Feasibility testing partitions the set of μDD into subsets of feasible and infeasible μDDs. It is possible for different μDDs representing different microarchitectural assumptions to be feasible. When this happens, experts can identify common structures in feasible μDDs to determine likely hardware features despite the ambiguity introduced by multiple feasible μDDs.

Consider Figure 2.6. There are four models, each identified by the presence or absence of features F_X and F_Y. Consistent models are highlighted in green and inconsistent models in red. There are two consistent models; introducing ambiguity in what model is the best fit. However, *all* consistent models contain Feature F_Y. If we have covered the relevant feature space—by running a wide enough range of programs to ensure that the hardware we are trying to reverse-engineer is adequately exercised—then CounterPoint can reliably conclude that Feature F_Y *must* be present. On the other hand, Feature F_X in isolation is insufficient to explain the performance counter observations, but it *is* possible that Feature F_X and Feature F_Y are both present. Given a feature space (*e.g.*, $F_X \times F_Y$), we can infer viable feature combinations.

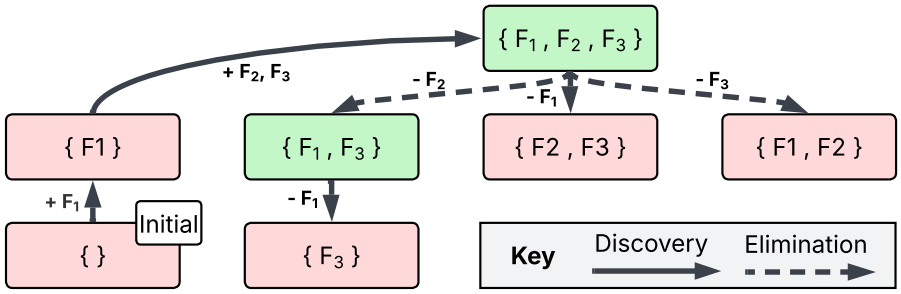


Figure 2.7: Expert-in-the-loop heuristic search algorithm navigates the model space without needing to explore the full cross-product of microarchitectural features. $F.$ denote microarchitectural features; models (boxes) are identified by (i) their set of features and (ii) their consistency with HECs.

Enumerating models. Feature discovery and model classification can be employed together to infer the presence of microarchitectural features. We propose a expert-in-the-loop algorithm for this purpose.

Our algorithm accepts an initial μDD and a dataset of HEC observations, and returns a set of μDD characterized by their features and feasibility. The algorithm consists of two phases: *discovery* and *elimination*. We advocate starting with a conservative model to ensure a more informative set of constraints that enable discrimination between candidate features, but the expert can start with any model. Features are discovered through the *Discovery* phase. Figure 2.7 shows an example search graph generated by the algorithm.

Discovery phase. Constraint violations are detected by CounterPoint, and the expert user eliminates the constraints by introducing new microarchitectural features or modifying existing ones. When more than one feature can eliminate a constraint, all features should be added to their model. This process is repeated until a feasible μDD is obtained.

In Figure 2.7, the initial μDD is shown at the bottom left. Feature F_1 is added to produce a new model $\{F_1\}$; features F_2 and F_3 are then added to create the μDD at the top of the tree $\{F_1, F_2, F_3\}$. This μDD is a candidate microarchitectural model for the system. At each iteration step, the model cones are verified to ensure that the model cone is expanded.

Elimination phase. The candidate μDD may contain more features than required for a feasible model. In the elimination phase, we recommend recursively pruning microarchitectural features until infeasible μDDs are obtained. This is based on our empirical observation–captured through

refinement of close to a hundred models–that pruning infeasible models tends to produce infeasible models, so the sub-tree need not be explored further. In Figure 2.7, features F_1-F_3 are removed from the top μDD to create separate μDDs. The μDD $\{F_1, F_3\}$ remains feasible, so features F_1 and F_3 are removed separately, resulting in infeasible μDDs.

2.5 Implementing CounterPoint

We implement CounterPoint as a Python library with roughly 3K lines of code, integrating with Pandas [110] for convenient data processing. To support broader community adoption, CounterPoint is designed for easy portability using a reproducible Docker [98] environment. We will share our MMU μDDs to help seed the development of improved MMU models in widely used software simulators [148, 87, 48, 47].

Domain-specific language for μDDs. We introduce a simple DSL for specifying μDDs: ACTION and COUNTER nodes are single-line statements, DONE nodes use the `done` keyword, and DECISION nodes are expressed with C-style switch cases. The DSL does not support functions, loops, or variables beyond μpath properties. Our DSL serves as a reference implementation, avoiding errors that could arise from deriving μDDs directly from RTL or C/C++ simulator specifications.

Feasibility testing. Given a μDD and a set of HEC observations, CounterPoint tests each observation for feasibility by constructing and solving a linear program (the complete linear program is provided in our technical report [86]). This entails enumerating every counter and μpath counter signature, implemented by a breadth-first traversal of the μDD. The back-end LP solver we use is `pulp` [101]. Constraint violations are identified by testing infeasible observations against the half-space defined by each constraint.

Deducing model constraints. Model constraints are derived from μpath counter signatures as follows. First, μpath counter signatures are normalized by dividing each element by the greatest common factor, and duplicates are removed. Second, Gaussian elimination identifies equality constraints and eliminates redundant HECs[6]. Third, μpath counter signatures that lie fully within the interior of the model cone are identified using linear programming and removed. Fourth, the

[6]For example, consider the following relationship:
`load.stlb_hit` = `load.stlb_hit_4k` + `load.stlb_hit_2m`.

conic hull is computed by: (i) adding the zero vector to the set of μpath counter signatures; (ii) computing the *convex hull*; (iii) selecting all faces which contain the origin, corresponding to the faces of the cone. The inequality model constraints are given by the planar equations of the resulting faces. We implemented this custom solution because no Python library computes conic hulls, and standard numeric methods (*e.g.*, QR factorization) are ill-conditioned, whilst symbolic operations preserve exact integer values.

2.6 A case study: The Intel Haswell MMU

We demonstrate CounterPoint's capabilities and evaluate its usability and performance on the Intel Haswell MMU. This case study shows how CounterPoint can uncover the behavior of advanced microarchitectural components, even when they interact deeply with complex systems software. The Haswell MMU is a strong case study target due to its rich set of HECs [66, 136] and its frequent use in prior research on address translation [150, 5, 85, 41, 9, 112]. Haswell also exhibits complex microarchitectural interactions across data and instruction activity [35], as well as under native and virtualized execution [22, 54, 4, 7, 23, 34, 111, 150, 117]. For these reasons, validating and/or refuting assumptions about the Haswell MMU represents a strong test of CounterPoint's effectiveness. This foundation positions us to extend our study to more modern architectures. For this study, we focus specifically on data-side activity in native execution. While full confirmation of our findings would require proprietary RTL, CounterPoint enables high confidence conclusions possible even without direct access to the RTL.

2.6.1 Guided model exploration.

Our initial model of the Intel Haswell MMU includes features that are well-established through documentation and prior research [24, 115, 41, 116, 56], and are typically integrated in software simulators. We assume a two-level TLB hierarchy and a four-level page table. Building on reverse-engineering studies of Haswell MMU caches [140, 50], we assume the presence of a PDE cache and an additional MMU cache for the page table level immediately preceding the PDE level. Consistent with conventional wisdom, we further assumed that the PDE cache is consulted once during every walk.

We refine the model using a diverse set of HEC observations from workloads that stress the MMU. We measured workloads from the GAPBS [20], SPEC2006 [62], PARSEC [31], and YCSB [38] benchmark suites, sweeping memory footprints from 250 MB to 600 GB using input generators. We also collected HEC data for two microbenchmarks: a linear access pattern (parametrized by footprint, stride, and load-store ratio) and a random access pattern (parametrized by footprint and load-store ratio). Through ablation studies, we found that removing these microbenchmarks causes us to miss violations of key model constraints (e.g., Constraint (1) in Table 2.1) that are essential for reverse-engineering the presence and trigger conditions of the TLB prefetchers described below[7]. To stress different MMU behaviors, experiments were repeated with 4 KB, 2 MB, and 1 GB page sizes. Together, these workloads and configuration options yield about 20 million HEC samples—enough observations to thoroughly stress-test our model assumptions and drive higher-quality model refinement.

We evaluate our models at the 99% confidence level. Across dozens of representative μDDs, we found that correlated counter confidence regions detect over 24% more model constraint violations compared to confidence regions that assume HECs are independent. For some models, exploiting correlations revealed over 75% additional violations compared to baseline. CounterPoint's confidence regions are effective because HECs are highly correlated: in our dataset we find that over 25% of counter pairs have a Pearson correlation coefficient that exceeds 0.9 (where 1.0 indicates perfect correlation, and 0.0 indicates no correlation).

With CounterPoint's support for guided model refinement, we explored dozens of μDDs. Our initial μDD contained 31 constraints, 8 of which were violated. We refined our initial μDD over several iterations, details of which we provide in our technical report[8] [86]. Across all explored models, there were thousands of μpaths and over a thousand model constraint violations. Our guided refinement surpasses prior ad hoc reverse-engineering efforts [12, 143, 50, 157], enabling us to uncover subtleties (with high confidence) in:

Address translation prefetchers. Several studies have proposed address translation prefetch-

[7] We ensured that all of our HEC measurements were unaffected by any published HEC errata. For errata that are triggered when SMT is enabled (*e.g.*, HSD29/HSM30 affecting `mem_uops_retired`), we addressed this by disabling SMT in the BIOS.

[8] Our Haswell MMU models continue to be expanded and refined. Our most recent results are available online in our technical report [86].

ing mechanisms, but little is known about how such prefetchers are actually implemented in real-world processors [127, 74, 30, 141, 97, 142]. Recent work suggests that underdocumented translation prefetching features may be at the core of unexplained performance anomalies in real-world workloads [21].

Using CounterPoint, we uncovered hardware in the Intel Haswell MMU that prefetches page table entries into its L1/L2 TLBs as well as PDE cache. Our analysis revealed three key aspects of the prefetcher's implementation:

First, we identified prefetch trigger conditions. If a workload is feasible with an μDD that includes the prefetcher but infeasible in one without it, the workload must trigger prefetches. This helps us deduce that prefetching logic scans virtual page numbers in the load/store queue and is triggered by sequential accesses predicted to cross a page boundary—contradicting the common assumption that prefetches are triggered exclusively by TLB misses. For increasing virtual addresses, prefetching is triggered after consecutive accesses to cache lines 51 and 52 within a page; for decreasing addresses, the trigger occurs at cache lines 8 and 7. No other cache line pairs were observed to initiate prefetching.

Second, we found that the load/store queue logic responsible for virtual-page prefetching relies on the page table walker to resolve translation prefetch requests. In practice, this means that prefetches trigger the walker to inject additional load instructions into the CPU pipeline—the same way it injects loads for demand page table walks themselves (previously called "ghost" or "stuffed" loads [157, 90]). In some cases, the walker generates hundreds of such additional loads. This overturns the prevailing model in prior work, which assumed prefetches bypass the pipeline and enter the memory hierarchy directly, and therefore model prefetches with unrealistically low latency. It also implies that significantly more prefetches can be injected than previously believed.

Third, we found that prefetch-induced page table walks abort when they encounter a page table entry whose access (reference) bit is unset—unlike regular page table walks, which set this bit. Consequently, the TLB prefetch does not complete. This behavior is logical: allowing a speculatively set access bit for an ineffective TLB prefetch could, in principle, lead to suboptimal paging decisions, and permitting TLB prefetches to set the access bit would also introduce additional microarchitectural complexity. Prefetch-induced page walks can still modify cache state, with

potential performance and security implications. Some recent TLB prefetcher proposals allow prefetch-induced page walks to set the access bit and complete [142, 141]. While this behavior is architecturally permitted [67], we have not observed it on Haswell.

Page table walk merging. Despite decades of research on TLBs and MMU caches, little is publicly known about how MMUs schedule page table walks. Using CounterPoint, we discover that MMUs can *merge* multiple outstanding walks to the same virtual page into a single page table walk, which we capture by modeling MSHRs within our MMU μDD.

Historically, MMU MSHRs have not been modeled in address translation studies because their design involves subtleties beyond those of conventional cache MSHRs [138, 81, 82]. For instance, page sizes—and therefore virtual page numbers—are unknown until after translation [41], making MSHR lookup and allocation non-trivial. Further, distinct page table walks have unique rules for updating access and dirty bits, as well as for determining whether they are allowed to touch physical memory regions marked speculative versus non-speculative [55]. These complexities make it far from obvious how walk merging can be safely implemented.

Our results show that MMU MSHRs are nonetheless critical for performance. For some workloads, page table walk merging reduces the number of distinct walks by nearly half. This finding underscores the importance of explicitly modeling MMU MSHRs in simulators used to evaluate address translation optimizations [29, 27, 14, 24, 18, 5, 12, 157, 76, 26].

Finally, CounterPoint reveals a surprising detail: the PDE cache is queried *before* outstanding walks to the same virtual page are merged. Prior studies have not considered this interaction [14, 24]. One might expect walk merging to reduce PDE cache lookups, easing port pressure, cutting bandwidth, and eliminating queuing delays. Instead, our μDD suggests that the PDE cache is looked up prior to MSHR allocation, likely to reduce latency via pipelining. Importantly, CounterPoint is able to do this because it enables discovery of not just individual hardware components, but also their relative placement within the pipeline.

Root-level MMU cache. A large body of address translation research proposing hardware optimizations [24, 29, 150, 125, 95, 130, 114] assumes the presence of a root-level MMU cache, yet some recent reverse-engineering studies have found no evidence of its existence [140, 50]. Coun-

terPoint demonstrates its compatibility with all other address translation features identified in this paper for the workloads we analyze, giving architecture researchers confidence in including it in their models. When *walk bypassing* is not modeled, several workloads become feasible only with a root-level MMU cache in the μDD. These workloads use 1GB pages which would stress a hypothetical PML4E cache (which is explicitly for 1GB pages), suggesting that for these workloads, the "missing" page table walker accesses could be explained by PML4E cache.

Aborted page table walks. Recent research has studied page table walks under speculation in modern out-of-order processors [85, 157, 58]. While prior work shows that x86-64 processors can abort in-flight page table walks in response to machine clears [85, 122, 157], the underlying implementation details remain poorly understood.

Using CounterPoint, we find that aborted walks are entirely consistent with all our HEC measurements and newly discovered features. Additionally, they appear to be triggered more frequently by workloads with high walker utilization. While more detailed study is necessary to better understand how aborted page table walks are implemented, CounterPoint suggests that page table walks can be aborted at any point—even before issuing a single memory access. This implies that aborted walks can still consume MMU and memory hierarchy resources, effectively imposing a hidden performance tax that should be explicitly modeled in simulation infrastructures for address translation [76, 5].

Page table walk replays. We observe that page table walks can complete without generating any memory accesses. This suggests that the core may include a mechanism allowing walks to finish without engaging the cache hierarchy. Prior work has shown a complex interplay between the hardware page table walker and microarchitectural structures that maintain memory consistency [157, 147]. We hypothesize that these "missing" accesses occur because walks are replayed or handled by hidden internal address translation caching structures not reflected in the `walk_ref` counter. Understanding these structures more concretely would require implementing new HECs or access to proprietary RTL. An alternative explanation is that the "missing" accesses do occur but are not counted because, unlike regular page-walker accesses, they are non-speculative. If we assume that aborted walks are replayed at micro-op retirement as non-speculative walks—as suggested in prior sources [55, 40, 147]—then the resulting μDD becomes feasible, but only once

features such as TLB prefetching and miss merging are incorporated.

2.7 Conclusion

We presented CounterPoint, a framework that transforms large HEC datasets into accurate, high-quality microarchitectural models. By encoding an expert's mental model as a μDD and automatically generating model constraints, CounterPoint eliminates the tedium and errors of manual derivation. At the same time, CounterPoint processes noisy HEC measurements into reliable, high-confidence ranges, bridging intuition with data-driven analysis. CounterPoint accelerates and sharpens the modeling of complex architectures, freeing experts to focus on insight rather than bookkeeping, and making advanced microarchitectural modeling faster and more insightful. By accelerating the productivity of these influential experts, insights extracted by CounterPoint's have the potential to shape the broader field of computing.

Chapter 3

Address translation hardware performance under increasing memory footprints

This chapter is adapted from "Understanding Address Translation Scaling Behaviours Using Hardware Performance Counters" which we published at IISWC'24. This paper was joint work with Abhishek Bhattacharjee.

3.1 Introduction

Virtual memory is an abstraction that simplifies programming. A key aspect of virtual memory is *address translation (AT)* which is implemented in hardware in the *memory management unit (MMU)*. In recent years, virtual memory has been established as a significant performance overhead in modern systems [25]. As a result numerous optimizations have been proposed [59, 56, 142, 75, 114, 131, 8, 58, 149, 28, 155, 120, 26, 123, 96].

To evaluate the effect of optimizations, architects frequently run studies (either real or simulated) with workloads that are known, or suspected to be, address translation intensive. Traditionally, architects have used the reference inputs provided with the benchmark suites to evaluate their designs [19, 132, 32]. However, many of the reference input sizes fail to generate large amounts of AT overhead, so architects generate new workload inputs that push the memory footprints into the giga- to tera-byte range.

Typically, one chooses an input size in a somewhat ad-hoc manner whilst considering multiple factors such as TLB miss rates [59] and the ease with which the corresponding memory footprint can be accommodated in the simulator [77]. However, the precise relationship between the input size and these various factors is not well understood. Understanding this is important for sim-

plifying the process of selecting appropriate workloads for experiments in the virtual memory research space. We thus devote this paper to teasing apart some of these relationships.

We define *address translation overhead* as the difference in runtime between one run of a program with address translation versus a run of the same program on a hypothetical system with no address translation cost. This is of interest to the architect because a large overhead indicates a large scope for potential performance improvement by optimizing the address translation stack.

$$\frac{\text{Walk cycles}}{\text{Instruction}} = \underbrace{\frac{\text{Accesses}}{\text{Instruction}}}_{\text{program}} \times \underbrace{\frac{\text{TLB misses}}{\text{Access}}}_{\text{TLB}} \times \underbrace{\frac{\text{PTW accesses}}{\text{PT walk}}}_{\text{MMU cache}} \times \underbrace{\frac{\text{Walk cycles}}{\text{PTW access}}}_{\text{cache hierarchy}}$$

Equation 1: Walk cycles per instruction equation. Brackets indicate the component that gives rise to each term.

Address translation overhead can only be approximated on real systems, so architects frequently use *proxy metrics* like TLB miss rate to predict it. We propose a new proxy metric called Walk Cycles Per Instruction (WCPI) which is defined as the ratio of the cycles spent performing page table walks against the number of instructions executed. This metric is appealing because it allows attribution of address translation pressure to individual sources as illustrated in Equation 1. Namely, WCPI can be expressed as the product of access intensity, TLB miss rate, page walker cache efficiency, and PTE lookup latency.

We use our definition of overhead and WCPI as a framework to help answer the following questions:

1. Do larger memory footprints lead to greater overheads?
2. How well does WCPI predict overhead?
3. At what memory footprints do different MMU components become bottlenecks?
4. How frequent are aborted and wrong-path walks?
5. How effective are 2MB pages in reducing overhead?

We reach the following conclusions (paper section indicated):

- For many AT intensive workloads, overhead scales logarithmically with memory footprint. *§3.5.1.*
- Walk cycles per instruction is a good proxy for AT overhead. *§3.5.2.*
- There is an apparent filtering effect where higher TLB hit rates can cause longer page table walks due to the MMU caches seeing less of the true access sequence. *§3.5.3.*
- Wrong path and aborted page table walks constitute a significant fraction of all walks. *§3.5.4.*
- The benefits of 2MB pages start to expire for workloads with footprints over 100GBs - within the range of footprints for modern-day workloads. *§3.5.5.*

The findings in this chapter were originally published [85] at IISWC'24, prior to the development of CounterPoint. We revisit these results in light of the Haswell MMU model derived using CounterPoint (Section 2.6), and find that the high-level conclusions remain largely accurate. In certain cases, we find that microarchitectural features identified by CounterPoint (like *TLB miss merging*), and our manual mapping of metrics to event counters, could cause misattribution of address translation overhead to individual MMU components, which we discuss in Section 3.6. We leave further refinement of the WCPI framework for future work.

3.2 Workload selection in address translation studies

Address translation overhead is typically only exposed by programs with large working sets since translations for smaller working sets can be (mostly) accommodated in the TLB. Historically, architects have selected programs from benchmark suites like SPEC [132] and PARSEC [32]. However, often the reference inputs for these workloads do not generate a great deal of address translation pressure, so architects generate **synthetic inputs** to drive up memory footprint in the hope of increasing AT pressure [56, 58, 114, 149, 155, 120, 26, 96] .

There has been work on creating input generators that can replicate the characteristics of known but proprietary inputs [83, 57, 100, 121]. Our work is orthogonal to these approaches and could be combined to create larger workload instances with similar characteristics.

3.3 Quantifying overhead

In this section we propose a definition for "address translation (AT) overhead" and discuss how to estimate it experimentally (Section 3.3.1). We justify our choice of baseline (Section 3.3.2). We introduce Walk Cycles per Instruction as a proxy for AT overhead (Section3.3.3).

3.3.1 Address translation overhead

We define the "address translation overhead" (AT overhead) of a workload to be the improvement in runtime that would be achieved in the absence of address translation (e.g. with 100% TLB hit rate and the absence of AT-related code). The overhead represents the maximal improvement in runtime that could be achieved if all address translation cost was removed.

We do not directly measure the AT overhead because it is impractical to eliminate every single TLB miss and bypass every line of AT-related code. Instead, we approximate the zero-overhead scenario by backing the workload with superpages. We run each workload with three page sizes: 4KB, 2MB, and 1GB[1]. We use `hugetlbfs` in combination with the *glibc malloc* `glibc.malloc.hugetlb` tunable to instruct *glibc* to back all malloc'd memory with the chosen page size (we do not change the page backing of non-heap segments.).

We select the smallest runtime from the 2MB and 1GB page sizes as the baseline runtime t_{baseline}:

$$t_{\text{baseline}} = \min(t_{\text{2MB}}, t_{\text{1GB}})$$

We define address translation overhead as:

$$\text{AT overhead} = t_{\text{4KB}} - t_{\text{baseline}}$$

Dividing this expression by t_{baseline} yields the relative AT overhead:

$$\text{relative AT overhead} = \frac{t_{\text{4KB}} - t_{\text{baseline}}}{t_{\text{baseline}}}$$

[1] `tc-urand` crashed for three input sizes when the page size was 1GB and so we exclude the 1GB performance counter data for `cc-urand` at those input sizes from our analysis.

3.3.2 Explanation of baseline

We use $\min(t_{2\text{MB}}, t_{1\text{GB}})$ as the baseline because we find that although performance with 1GB pages is usually better or similar to that of 2MB pages, it can occasionally be worse. In particular, this happens at small memory footprints (for our workloads up to 20GB total memory usage) because the memory allocator cannot back regions smaller than 1GB in size with 1GB pages, causing these regions to instead be backed by smaller pages. However, at smaller memory footprints, 2MB pages eliminate most translation overhead, so we would expect the difference between the true overhead to be minor. At larger footprints, the performance with 1GB pages is usually better or similar to the performance with 2MB pages and so the 1GB runtime is selected.

3.3.3 Walk cycles per instruction

We present *walk cycles per instruction* (WCPI) as a measure of pressure on the address translation stack. We define WCPI as the *ratio of total page walk cycles to total instructions executed.* Unlike AT overhead, it can be calculated from results for a single run of an experiment.

Equation 1 expresses the relationship between WCPI and other measures of AT pressure. Each component is labeled by the component that gives rise to the term. Hence, the WCPI equation is a useful reference for understanding the key relationships between the various components of the address translation stack: namely the TLB, MMU caches, and caching of PTEs in the cache hierarchy.

Note that walk cycles per instruction is not the same as the translation latency per instruction. This is because the system we test (and most state-of-the-art processors) feature multiple levels of TLBs with varying lookup latencies, so even the latency of a TLB hit is variable - an effect not captured in the WCPI equation. Unfortunately, the performance counters on our system do not provide enough information to be able to differentiate between L1 and L2 TLB hits for retired instructions, which are of interest to us.

Despite this, we argue that the WCPI is sufficient for the following reasons. Firstly, the latency difference between an L1 and L2 TLB hit (8 cycles on our system [1]) is much smaller than the latency of a page table walk. Secondly, it is easier to hide the latency of an L2 TLB hit than it is

Table 3.1: Workloads
(ST = single-threaded, MT = multithreaded)

Suite	Program	Generators	Type
gapbs [19]	bc, bfs, cc, pr, tc	urand, kron	graph processing (MT)
ycsb [39]	memcached	uniform	key-value store (MT)
spec2006 [132]	mcf	rand	network simplex (ST)
parsec [32]	streamcluster	rand	clustering (MT)

Table 3.2: Input Generators

Generator	Descriptions
urand	uniform random graph
kron	scale-free graph
uniform	95% read, 5% write requests, uniform distribution. 67108864 records of 8KB
(mcf) rand	N timetabled trips and $N(N-1)$ deadhead trips
(streamcluster) rand	list of random vectors

an L2 miss followed by a page table walk, so we believe that L2 hits will have a less significant impact on performance. Finally, we show that WCPI is strongly correlated with AT overhead in Section 3.5.2.

3.4 Methodology

We select a range of programs that are typically considered address translation intensive (Table 3.1). These include graph processing, network simplex, key-value stores, and clustering algorithms. We use the corresponding input generators listed in Table 3.2. `urand`, `kron`, `uniform`, and `(streamcluster) rand` are embedded in their benchmark suites. We wrote the `rand` generator for `mcf` ourselves.

We call the combination of program and input generator a *workload*. We denote a workload as `program-inputgenerator`, unless the program only has one input generator (e.g. mcf and streamcluster), in which case we may simply refer to it by its program name (e.g. `mcf`). For each workload, we sweep the input sizes to generate program instances with memory footprints in the $\sim$ 250MB to $\sim$ 600GB range. For the remainder of this paper, we refer to each input size by its

Table 3.3: System

Component	Description
CPU	2 sockets × 6c Intel Xeon E5-2680 v3 @ 2.5GHz 32KB L1D, 256KB L2 cache per core 30MB shared L3 shared cache per socket TLB-L1D: 64×4KB, 32×2MB, 4×1GB TLB-L2: 1024× shared 4KB/2MB pages 1 page table walker
DRAM	384GB ECC DDR4 @ 1600 MHz per socket (2)
OS	Linux Kernel 6.5.0-25-generic

corresponding memory footprint in the 4KB configuration since this is a more tangible quantity.

Table 3.3 describes the system we used in our experiments. Our workloads can be long-running (up to 3 days), so we run experiments in parallel across three identically configured systems. To avoid potential sources of systematic error, we run all input size sweeps for a particular workload on the same machine.

We follow best practices to maximize performance and reduce sources of noise in our experiments by: disabling simultaneous multithreading (SMT); disabling dynamic frequency and voltage scaling (DFVS); limiting co-running applications to OS services only; giving our workloads high scheduler priority; disabling address space layout randomization (ASLR); and warming up file system caches with a 60 second dry run of the experiment.

3.5 Results and analysis

3.5.1 Do larger footprints lead to greater overheads?

We would expect in general that workloads with larger memory footprints would have greater AT overhead. We test this hypothesis by plotting relative AT overhead against the measured memory footprint in Figure 3.1.

Inter-workload. There is a positive correlation between footprint and relative AT overhead. However, there is a large degree of variation that arises because different workloads have different access patterns and different levels of performance sensitivity. We discuss this on a per-workload basis next.

Intra-workload. When we consider each workload individually, we find that there is a strong correlation between memory footprint and AT overhead. For this discussion, we will use `cc-urand` as an illustrative example (Figure 3.2). Visually, we can see that there is a linear relationship between relative AT overhead and the *logarithm* of the memory footprint. In other words, it appears that:

$$\text{relative AT overhead} \approx \beta_m \log_{10}(m) + \beta_0$$

where M is the memory footprint and β_0, β_M are constants.

At first, this may seem surprising - why should the overhead grow with the $\log$ of the memory footprint? Our results agree with earlier work [73] by Jurkiewicz and Mehlhorn, who demonstrated that workloads with certain types of access patterns have an additional $\log x$ asymptotic scaling component due to the presence of virtual memory. This overhead arises because the page table is implemented as a radix *tree* which must be walked on a TLB miss. Whilst the TLB and MMU caches can hide some of the overhead, they do not affect the asymptotic behavior. What is perhaps surprising is that this $\log$ scaling behavior is observed even though the page table only has four levels (which we might otherwise consider to be so few that the system is far from the asymptotic regime).

We calculate the scaling coefficients on a per-workload basis by linear regression against $\log_{10} M$ and a constant. The results are presented in Table 3.4.

We find that for most workloads there is a strong linear correlation as indicated by the high adjusted R^2 values. Additionally, we find that the average coefficient of the $\log M$ term is 0.13 for all workloads with a strong linear correlation (where $R^2 > 0.9$). This means that an increase of $10\times$ in the memory footprint causes a 13% increase in AT overhead.

There are four exceptions to this trend: `mcf-rand`, `memcached-uniform`, `streamcluster-rand`, and `tc-kron`. We plot their trends in Figure 3.3 and discuss them here.

With `mcf-rand`, the relationship is highly nonlinear. At smaller footprints, AT overhead grows slowly, before exploding at larger footprints. In fact, AT overhead grows fastest for `mcf-rand` out of all the workloads.

`memcached-uniform` is a key-value caching workload and exhibits complex scaling behavior

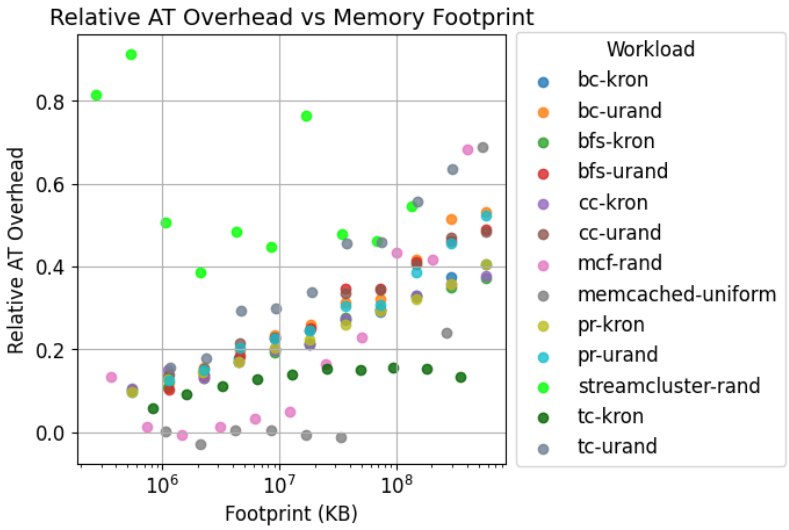


Figure 3.1: Relationship between relative AT overhead and memory footprint, grouped by workload.

because the key-value cache hit rate varies with the memory footprint. At small memory footprints, performance is insensitive to the page size. Increasing the memory footprint ultimately does cause an increase in overhead, albeit in a nonlinear fashion.

For `streamcluster-rand`, there is a lot of variation. There is no evidence of any clear pattern, which would suggest the AT overhead is more strongly determined by factors other than the memory footprint.

Finally for `tc-kron` the overhead increases but levels off. We speculate that this is because the `tc` algorithm contains an optimization for handling scale-free graphs (like those generated by `kron`) [19]. This leads to an effective access pattern that scales in a friendly manner from an address-translation perspective. Despite overhead not strongly increasing with footprint, the overhead is still large (up to $\approx 15\%$) suggesting that this workload could still benefit from address translation performance improvements.

Conclusion: In general, the greater the memory footprint the greater the relative AT overhead. However, there are exceptions - even for workloads that are traditionally thought of as being "address translation intensive".

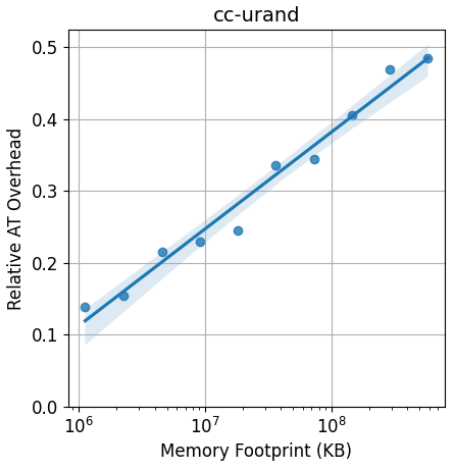


Figure 3.2: Relative AT overhead vs memory footprint for `cc-urand`

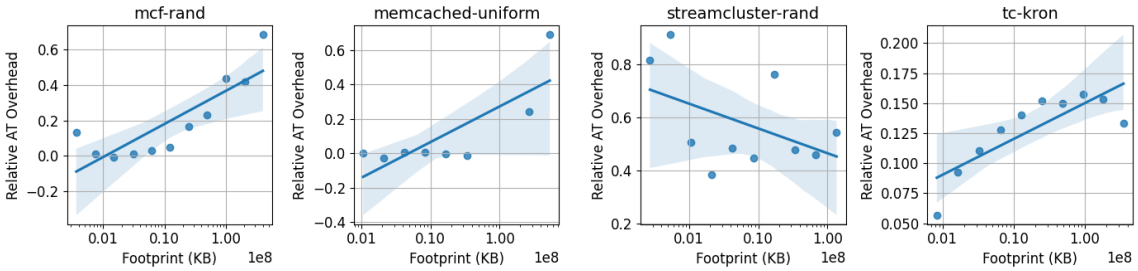


Figure 3.3: Relative AT overhead vs memory footprint for workloads with weaker linear correlations.

Table 3.4: Regression results for model relative AT overhead $= \beta_0 + \beta_1 \log_{10} M + \epsilon$.

		Coefficients		Statistic
		const	$\log_{10} M$	Adj. R^2
Program	Generator			
bc	kron	-0.497	0.101	0.982
	urand	-0.830	0.153	0.959
bfs	kron	-0.471	0.097	0.986
	urand	-0.797	0.147	0.987
cc	kron	-0.442	0.093	0.974
	urand	-0.695	0.135	0.973
mcf	rand	-1.129	0.187	0.667
memcached	uniform	-1.381	0.207	0.580
pr	kron	-0.479	0.099	0.990
	urand	-0.739	0.139	0.956
streamcluster	rand	1.215	-0.094	0.122
tc	kron	-0.089	0.030	0.627
	urand	-1.048	0.196	0.970

3.5.2 How well does WCPI predict overhead?

In this section we evaluate the suitability of using walk cycles per instruction (WCPI) as a proxy for address translation overhead. We compare WCPI against four other metrics: TLB miss rate, TLB misses per instruction, the fraction of clock cycles with an outstanding page table walk, and the walk cycles per access.

We evaluate the relationship between the metric and relative AT overhead with two statistics: the *Pearson correlation coefficient* and the *Spearman rank correlation*.

The Pearson correlation coefficient describes the degree of linear correlation between the metric and relative AT overhead. The magnitude describes the extent of linear correlation and the sign describes the direction. The maximum magnitude is one and this indicates a perfect linear correlation. We include the Pearson correlation coefficient because it helps us quantify the degree of linearity between the AT pressure metric and the overhead, which is useful if one wishes to model the relationship.

The Spearman rank correlation coefficient is another measure of similarity that operates on the difference in ranking order between the metric and relative AT overhead. That is, a Spearman rank correlation close to 1 indicates that the order of the workloads when ranked by the metric is similar to the order when ranked by relative AT overhead. It is less strict than the Pearson correlation in the sense that it measures how "monotonic" the relationship between two variables is, instead of the degree of linearity. We include this measure because it reflects the approach one might take when picking workloads (e.g. pick the ten workloads with the most AT pressure.) We repeat this analysis both across all workloads (inter-workload), and within a single workload whilst sweeping the input size (intra-workload).

There are 4 (out of 132) workload-input size combinations where the relative AT overhead was measured to be negative[2]. We consider these workload-input size combinations not to be AT sensitive and exclude them from the regression analysis. The results are not excluded from the rest of the paper.

Inter-workload. Table 3.5 shows the various correlation coefficients between AT pressure

[2]One input belongs to `mcf` and the other three belong to `memcached`.

Table 3.5: Strength of correlations between metric and relative AT overhead.

Correlation coefficient AT pressure metric	Pearson	Spearman's rank
TLB misses per kilo access	0.452	0.582
TLB misses per kilo instruction	0.364	0.579
Walk cycle fraction	0.555	0.688
Walk cycles per access	0.462	0.769
Walk cycles per instruction	0.567	0.768

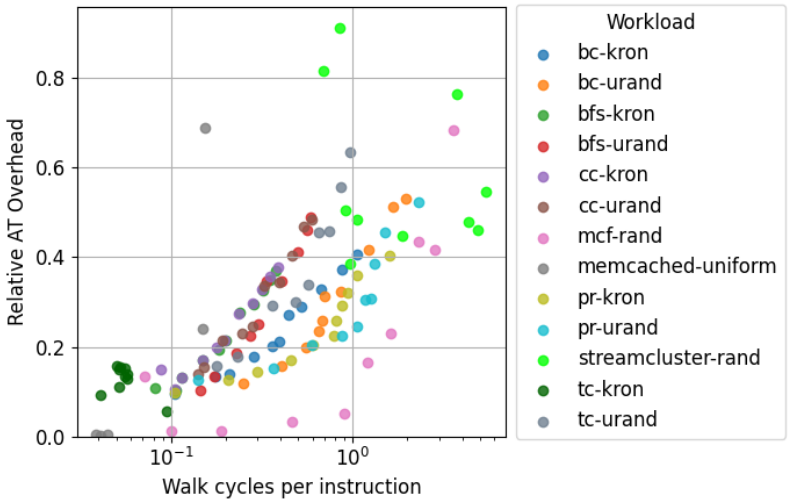


Figure 3.4: Relationship between relative AT overhead and walk cycles per instruction, grouped by workload. Workload-input size combinations with a negative measured relative AT overhead are assumed to not be AT sensitive and are excluded from the figure.

and AT overhead when we consider all workloads and memory footprints together. We see that TLB misses per kilo instruction performs the worst in both measures, whereas WCPI performs best in Pearson correlation and a close second-best Spearman's rank. Although WCPI has a reasonable Pearson correlation coefficient, it is still significantly less than one.

To understand this, we plot the relationship between WCPI and AT overhead in Figure 3.4. The Pearson correlation coefficient is far from one due to multiple sources of nonlinearity. Firstly, there are nonlinearities arising from different workloads having fundamentally different characteristics. But even within a workload, there is nonlinearity, because overhead does not necessarily scale linearly with walk cycles.

Intra-workload. We now consider the correlation between WCPI and overhead within a

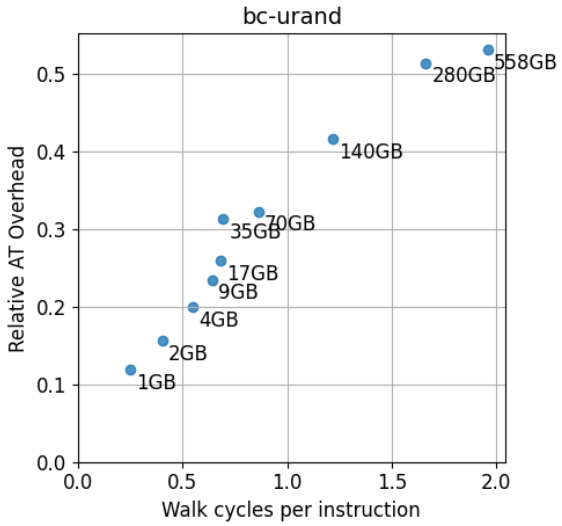


Figure 3.5: Relationship between AT overhead and WCPI for `bc-urand`. Each point is labeled by memory footprint.

single workload. We select `bc-urand` as a representative workload and plot the overhead-WCPI relationship in Figure 3.5. We make the following two observations:

Firstly, there is a monotonically increasing relationship between WCPI and overhead. This aligns with our intuition that the greater the walk cycles the greater the overhead. It indicates that the increase in latency cannot be hidden by the out-of-order processor and hence contributes to the critical path of execution.

Secondly, the relationship is nonlinear, which we attribute to the dynamics of the program changing as it scales. By dynamics, we refer to the composition of the dynamic instruction stream (e.g. instruction types and dependencies). This can change as the input size varies; for example, see the second row of graphs in Figure 3.6, which illustrates how accesses per instruction varies with memory footprint. As another example, consider `memcached`, which is a key-value (KV) cache. When the memory footprint is small, the KV cache hit rate is small and most of the code handles cache misses. But as we scale up the footprint, the code distribution changes so that the hit path is exercised more frequently. The basic block execution distribution can be very sensitive to input size; what is more, different basic blocks may be able to hide memory latency by different amounts. All of these factors introduce nonlinearity.

The degree of monotonicity between WCPI and relative AT overhead can be quantified by

the Spearman rank correlation coefficient. Seven workloads have a coefficient of 1.0 exactly; three workloads have a coefficent between 0.9 and 1.0; and three workloads have coefficients less than 0.9: `mcf-urand`, `streamcluster-rand`, and `cc-kron`. We examined those three mentioned workloads graphically and we found that WCPI appeared almost totally uncorrelated with relative AT overhead.

Conclusion: When comparing across all AT-sensitive workloads and input sizes, the strongest Pearson correlation and near-strongest Spearman Rank correlation occurs between *walk cycles per instruction* and *relative AT overhead*. For most workloads, there is a non-decreasing monotonic relationship between WCPI and relative AT overhead.

3.5.3 When do different MMU components become bottlenecks?

There are multiple components of the memory management unit that contribute to AT pressure: the TLB, the MMU caches, the page table walkers, and the cache hierarchy. Whilst we expect pressure on each component to become worse with memory footprint, it is less clear at what memory footprints each of these components become bottlenecks. To understand this better, we plot all components of the WCPI equation (Equation 1) in Figure 3.6. Due to space limitations, we select the following four workloads, although these illustrate effects that we see replicated across the whole span of workloads: `bfs-urand`, `mcf-rand`, `pr-kron`, and `tc-kron`.

We now discuss each WCPI equation component in turn.

Walk cycles per instruction. For three out of the four workloads, WCPI increases monotonically with input size. This is intuitively what we would expect for our workloads since they are known to be translation-intensive. We can also see that for the three workloads with this scaling behavior, to a first-order approximation, the WCPI grows with the log of the memory footprint - much like the overhead. The one exception to this rule is `tc-kron`, whose WCPI remains both low and relatively flat. We hypothesize that this is due to its aforementioned graceful scaling behavior.

To understand the trends in WCPI, we deep dive into the scaling behavior of each of the individual components (and their interactions).

Regular accesses per instruction. For `bfs-urand`, `mcf-rand` and `pr-kron`, the accesses per

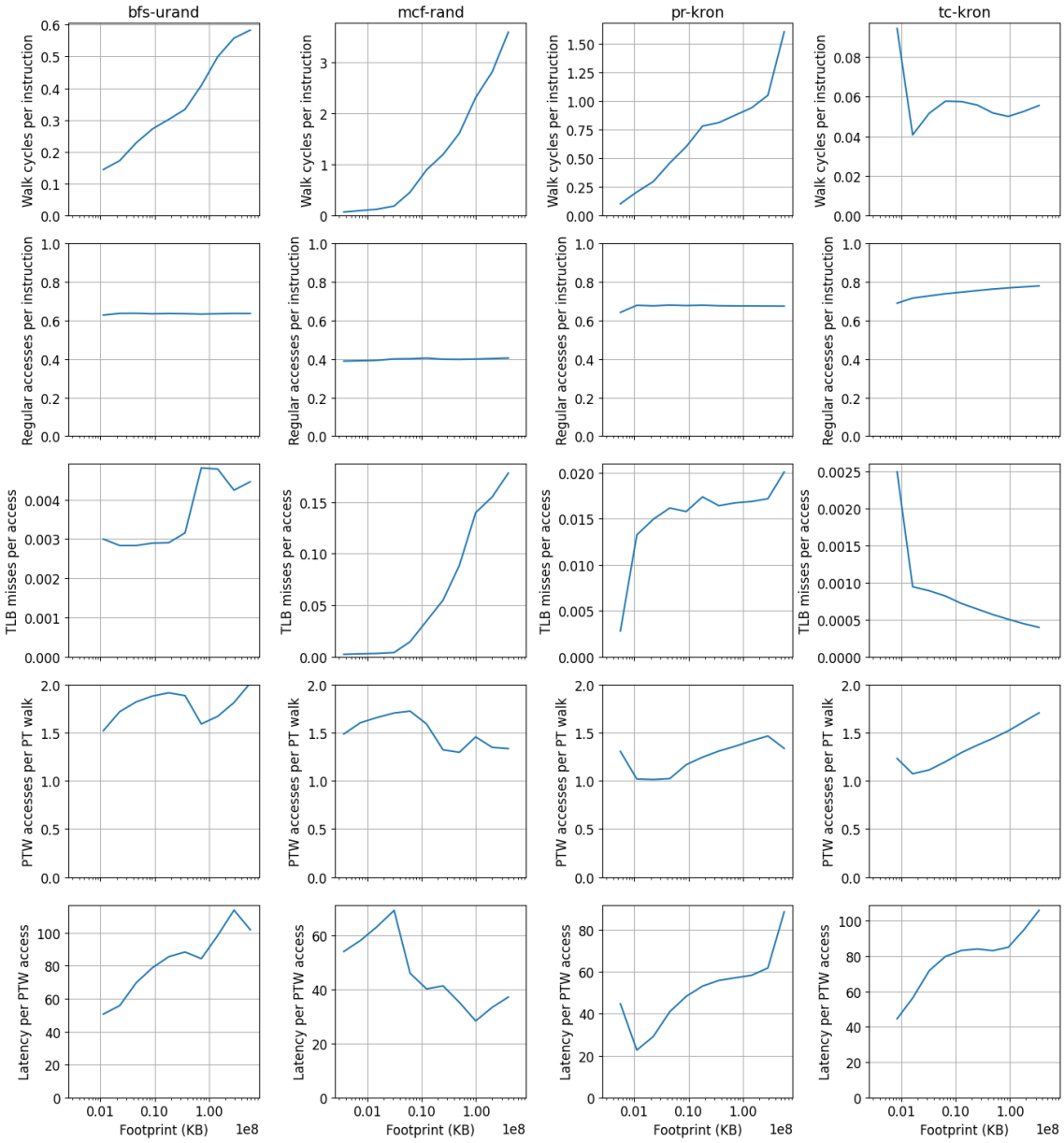


Figure 3.6: Component-wise breakdown of the scaling behavior of four workloads.

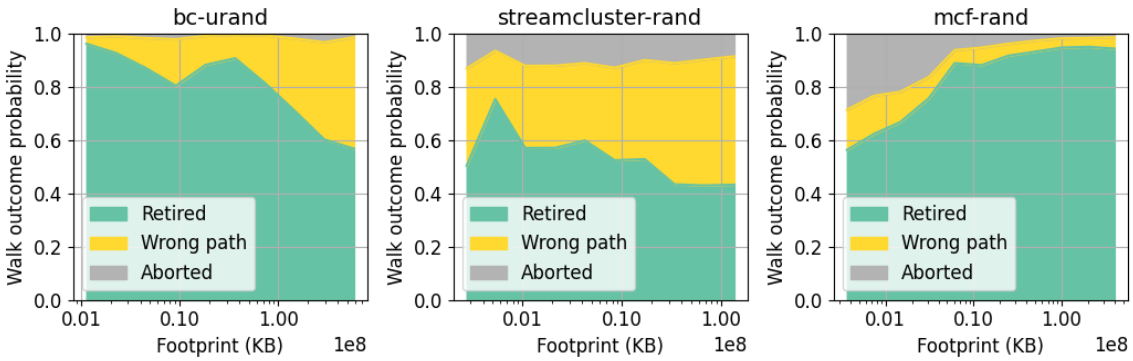


Figure 3.7: Walk outcome distribution as a function of memory footprint. The vertical width of each band corresponds to the probability that any initiated walk has that outcome.

instruction remains stable which provides evidence that the program has reached some limiting behavior in terms of instruction composition. However, this is not necessarily true in general. The instruction mix for `tc-kron` changes across the whole memory footprint range, which suggests that the dynamics of `tc-kron` changes somewhat significantly with input size.

TLB misses per access. We originally hypothesized that the TLB miss curve would be sigmoidal in nature. However, there is no real sigmoidal trend for the workloads and memory footprint ranges we consider. Both `bfs-urand` and `pr-kron` appear to have well-defined cliffs, which we would expect to see when we have a big jump in working set size. However, with `mcf-rand` the TLB miss rate increases with no clear sign of leveling off - which suggests that `mcf-rand` must be pushed past the 380GB footprint to saturate. `mcf` is also noticeable for its very high TLB miss rates - at the largest footprint we studied around 20% of accesses result in TLB misses. `tc-kron`, again as the exception, actually has a decreasing TLB miss rate.

Accesses per walk. The average number of accesses per page table walk depends on two factors: MMU cache effectiveness and aborted page table walks. We assume that the trends we see here are due to MMU cache effectiveness; we discuss aborted page table walks more in Section 3.5.4.

In general, the number of accesses per page table walk lies within 1 and 2, across all memory footprints. This indicates that the page walk caches are generally doing a good job.

Out of all the metrics, the number of accesses per walk is the most unpredictable. This is not surprising. The CPU we used in our experiments likely has at least two levels of page table walk

Table 3.6: Walk outcome metric formulae.

Outcome	Formula
Initiated	`load.causes_walk` + `store.causes_walk`
Completed	`load.walk_done` + `store.walk_done`
Retired	`load.ret_stlb_miss` + `store.ret_stlb_miss`
Aborted	Initiated - Completed
Wrong path	Completed - Retired

caches [139], each of different sizes; so when the spatial-locality pattern of a workload changes the complex interplay between the various MMU caches also changes.

We observe that a decrease in the number of accesses per page table walk often occurs with an increase in the TLB miss rate (all workloads except `tc-kron`). We propose this is analogous to the "filtering effect" that affects conventional multi-level caches [146, 36, 70]. As the TLB miss rate increases, the filtering effect of accesses is reduced. This enables the MMU caches to "see" more of the true virtual memory access pattern leading to better MMU cache replacement and hence fewer accesses per walk.

Latency per walk access. The latency per walk access is a function of the hotness of PTEs in the cache hierarchy.

For most workloads, the latency per walk access increases with memory footprint. This is consistent with what we would expect, because at larger footprints there is greater cache contention (both between PTEs and between PTEs and regular data), and hence the PTEs would become colder in the cache hierarchy.

We plot the PTEs access location distribution against memory footprint in Figure 3.8. We use `pr-kron` as an illustrative example. At the smallest footprints, most of the PTEs are found in the L1 and L2 caches. This jumps up to around 90% of accesses around 10^6 KB. This is accompanied by a large increase in TLB miss rate, suggesting the PTEs become hotter in the cache hierarchy because they are no longer filtered by the TLB. Increasing the footprint further results in the PTEs moving further away from the core, as indicated by the increasing fraction that hits in the L3 cache and memory. A small but non-negligible fraction are found in memory at the largest memory footprints. Despite being a small fraction of all accesses, this significantly drives up the

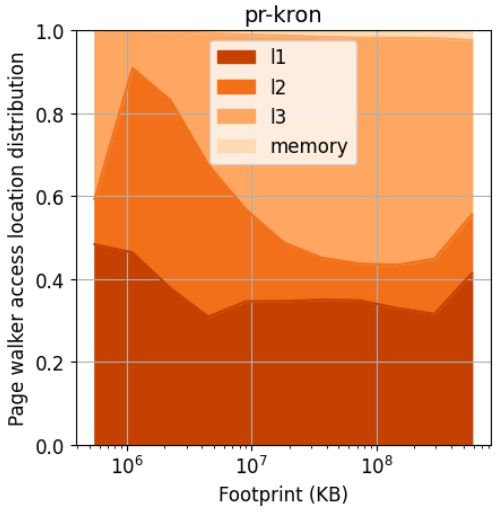


Figure 3.8: Distribution of PTE access location as a function of input size for `pr-kron`. The vertical width of each band is the probability that the page table walker finds the PTE in that location.

average latency because the latency of a memory access is so huge.

`mcf-rand` is an exception to "larger footprints means colder PTEs". The average latency per walk access decreases with increasing memory footprints. We speculate this is because `mcf` has a high TLB miss rate that grows to $\approx 18\%$. Because of this, PTEs increasingly displace regular data in the caches closer to the core, and hence the average access latency decreases. In some sense, PTEs "outcompete" regular data. We hypothesize that this does not occur with the other workloads because their smaller TLB miss rates mean that regular accesses continuously apply a force that pushes PTEs toward memory.

Conclusion: As we hypothesized, there is no "one-size-fits-all" explanation for why address translation pressure increases with footprint. It is the product of multiple interacting factors including the TLB miss rate, page walk cache effectiveness, and hotness of PTEs in the cache hierarchy. We speculate that WCPI is the pressure metric most strongly correlated with AT overhead because it implicitly captures all the individual terms *and* their interactions.

3.5.4 How frequent are aborted and wrong-path walks?

All initiated pages table walks have one of three outcomes: they can correspond to a **retired** instruction; they can complete on a speculated wrong path (**wrong path**), or they can be **aborted**

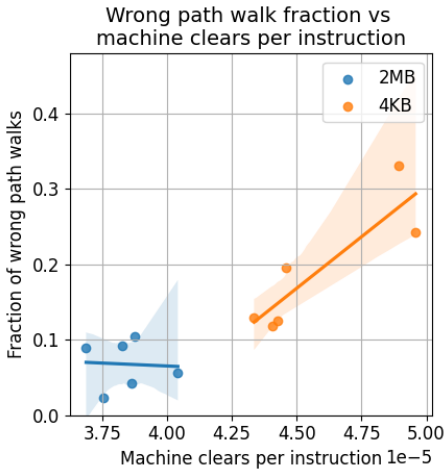


Figure 3.9: Wrong path walk fraction vs machine clears per instruction.

before they are finished. Table 3.6 shows how we compute counts from performance counters.

We plot the outcomes of page table walks for 3 workloads in Figure 3.7. Most workloads have behavior similar to `bc-urand`. At low footprints, the fraction of aborted/wrong-path walks is small: around 10% combined. Surprisingly, as the workload scales, the fraction of wrong-path walks increases significantly - a behavior we see for most workloads. At large footprints this can become dramatic: with `bc-urand` almost 50% of all initiated walks are either wrong-path or aborted.

For `bc-kron` we collect additional performance counters including branch mispredictions and machine clears. We were unable to find any clear relationship between the branch misprediction rate and the fraction of wrong path walks. We plot the non-correct path walk fraction against the number of machine clears per instruction in Figure 3.9. We see that generally, an increase in machine clears per instruction is associated with an increase in the combined fraction of misspeculated and wrong-path walks. Machine clears have multiple causes including incorrect memory dependence prediction and memory ordering violations. More study is needed to understand why these arise more with 4KB pages.

`streamcluster` and `mcf` are noticeable because they exhibit a large fraction of wrong-path and aborted walks even at smaller memory footprints. With `streamcluster`, the problem becomes worse with input size (up to 57% wrong-path or aborted walks); whereas for `mcf` the fraction of wrong-path and aborted walks reduces.

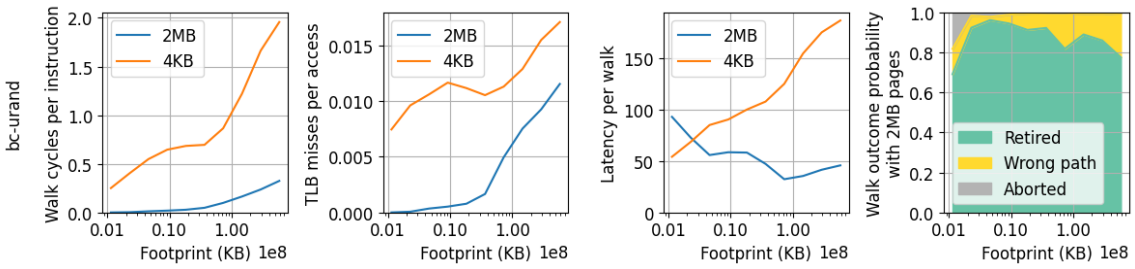


Figure 3.10: Key address translation metrics for `bc-urand` with 2MB pages. Comparison with 4KB pages included for all graphs except walk outcomes.

Conclusion: Aborted and wrong-path walks constitute a significant fraction of all initiated page table walks. For most workloads, the fraction increases with increasing input size.

3.5.5 How effective are 2MB pages?

2MB superpages have, until recently, been considered a silver bullet for improving address translation performance. However, recent work has identified that further improvement is possible by using 1GB huge pages [123], implying that 2MB pages are not optimal.

As a representative example, we plot key address translation metrics with 2MB superpages for `bc-urand` in Figure 3.10. For comparison, we also include the metrics with 4KB pages when possible.

Looking at the WCPI curve, we see that 2MB superpages result in significantly less AT pressure, as expected. This is due to both the TLB miss rate being lower and the average page walk latency being shorter. However, observe at very large footprints that the TLB miss rate starts rising dramatically (around the same time that the WCPI starts increasing). It seems likely that further increasing footprint will further increase WCPI.

The average walk latency appears somewhat flat with 2MB superpages. The average walk latency is the product of the number of accesses per walk and the average latency per access, both of which are lower for 2MB superpages at all but the lowest footprints (plots not shown due to space constraints). However, we do not expect this trend to last: a workload with a 4TB memory requirement would require 16MB of 2MB page table entries, a quantity likely large enough to cause significant cache pressure (and hence make PTEs colder in the cache hierarchy).

Finally, we plot how the walk outcome distribution changes with memory footprint for 2MB

pages. While wrong-path and aborted walks are still significant at the largest footprints ($\approx 20\%$), it remains much less than the 4KB case which we showed in Figure 3.7. This suggests that huge pages have effects beyond simply reducing walk latency and the TLB miss rate.

Conclusion: Although 2MB superpages significantly reduce AT pressure, the trend in walk cycles per instruction at very large memory footprints suggests that the magnitude of this benefit does not continue to grow for ever larger memory footprints. However, 2MB pages do reduce the number of aborted and wrong-path page table walks.

3.6 Error analysis with CounterPoint

The measurements in this study were made on a real Intel Haswell microprocessor using hardware event counters. Our methodology required expressing each term of the walk cycles per instruction formula (Equation 1) as a function of event counter values, as we similarily did for the walk outcome metric formula (Table 3.6). We performed this mapping manually, using existing hardware event counter documentation [66] for reference

Subsequent to the publication [85] of this work, we invented CounterPoint. Using CounterPoint, we discovered that the Haswell MMU likely exhibits more complex behaviors that were previously assumed (Section 2.6). These more complex behaviors could influence the results of this study, because they may be captured by the hardware event counters we chose to measure certain microarchitectural quantities (*e.g.* TLB misses, retired walks).

We used our CounterPoint-derived Haswell MMU model to identify possible sources of errors in our microarchitectural metrics. First, we used our final Haswell μpath Decision Diagram to identify the behaviors captured by each event counter. We then tracked how each behavior aggregates across each event counter formula. We compared this aggregated behavior against the original intent of the formula to identify sources of error, which we document in Table 3.7. Assuming our Haswell MMU model is accurate, we find that many terms are vulnerable to being either underestimated ('$-$' column) or overestimated ('$+$' column). The exact estimation error sources are listed in Table 3.8. In general, we find that these errors occur because the microarchitecture exhibits more complex behaviors than were originally considered (E1-3,E5), and/or

Table 3.7: Every term in the formulae presented in this paper was manually mapped to Intel Haswell hardware event counters. Inspecting the event counter formula against the Haswell MMU model derived with CounterPoint reveals possible sources of error. '+' indicate sources of over-estimation and '−' lists sources of under-estimation. 'Unknown' cannot be determined from the CounterPoint model. This is because the event counter formula includes latency terms which are not currently supported by CounterPoint (`load.walk_dur` and `store.walk_dur`), or because the event (`ins`) is not included in the current model.

Abstract Term	**Event Counter Formula**	+	−
Walk cycles	`load.walk_dur` + `store.walk_dur`	Unknown	
PTW accesses	`walk_ref.l1` + `walk_ref.l2` + `walk_ref.l3` + `walk_ref.mem`	E1	E2
TLB misses	`load.walk_done` + `store.walk_done`		E4,E5
Page table walks	`load.walk_done` + `store.walk_done`		E4
Accesses	`load.ret` + `store.ret`		E6
Instructions	`ins`	Unknown	
Retired walks	`load.ret_stlb_miss` + `store.ret_stlb_miss`	E5	E3
Aborted walks	`load.causes_walk` − `load.walk_done` + `store.causes_walk` − `store.walk_done`		
Wrong path walks	`load.walk_done` − `load.ret_stlb_miss` + `store.walk_done` − `store.ret_stlb_miss`	E3	E4, E5

Table 3.8: Microarchitectural behaviors captured by hardware event counters that are not explicitly considered in the address translation scaling analysis. The behaviors are present in the Haswell MMU model derived using CounterPoint, and indicate potential sources of error.

	Error Source
E1	`walk_ref` includes accesses made by prefetch-initiated walks.
E2	Replayed page table walks do not increment `walk_ref` events.
E3	A demand initiated page table walk followed by a replayed walk only increments `load.ret_stlb_miss` or `store.ret_stlb_miss` once.
E4	Initiated page table walks that are unsuccessful or aborted are not counted by `load.walk_done` or `store.walk_done`.
E5	Multiple TLB misses may be merged into a single page table walk.
E6	Not all demand memory access micro-ops retire.

because the event counters do not align with the desired quantity (E1-4, E6)

We do not believe that all of these errors are likely to be significant. Our workloads are largely random access and therefore unlikely to trigger the sequential access TLB prefetcher, making E1 likely negligible. We expect the number of replayed page table walks to be a small, making errors E2 and E3 likely insignificant. The most likely sources of error are aborted walks (E4), TLB miss merging (E5), and squashed memory accesses micro-ops (E6). Consequently, we likely overestimate *PT accesses per PT walk*, and potentially either under- or over- estimate *TLB misses per access*, depending on the workload characteristics. We were unable to determine possible error sources in *walk cycles* or *instructions* due to the current limitations of CounterPoint (*i.e.* no support for latency counters), and of our Haswell model (*i.e.* no `ins` counter). However, the strong correlation between *walk cycles per instruction* (WCPI) and address translation overhead gives us confidence that the WCPI metric is accurate.

3.7 Discussion

In this work we set out to understand the complex interplay between program, input, address translation pressure, and address translation overhead using a novel analytical model. In the process, we have made many insights. We summarize them here and provide suggestions as to how they might guide further research in address translation:

Overhead usually scales with the log of the memory footprint The relative address translation overhead tends to scale with the log of the memory footprint for our workloads. This is consistent with earlier theoretical work which showed that the radix tree data structure interacts with caches to produce this behavior. *Alternative page table data structures that do not introduce a* $\log M$ *overhead are deserving of further study.*

Larger footprint does not always mean larger overhead. Despite the general trend of workloads with larger memory footprints having larger address translation overhead, this is not always the case - even for workloads that are thought of as being address translation intensive (e.g. `tc-kron`). *Practitioners should consider empirically verifying the scaling behavior of their workload.*

Walk cycles per instruction is a good proxy for AT overhead. The best way to find the

AT overhead for a workload is by measuring it directly. Unfortunately, this involves re-running the workload in a controlled environment, which is not always possible (e.g. when gathering data from production systems). We found that walk cycles per instruction (WCPI) is a good proxy measure for AT overhead. *Consequently, we believe that using WCPI as a heuristic to guide huge page allocation either in the compiler or operating system would be worthy of further investigation.*

Higher TLB hit rates can cause longer page table walks. We see in our data that an increase in the TLB miss rate is often accompanied by an increase in MMU cache effectiveness and an increase in the hotness of PTEs in the cache hierarchy. We hypothesize that this is analogous to the filtering effect observed with multi-level caches, where exposing outer caches to more of the underlying access pattern can improve hit rates [146, 36, 70]. *Further research to quantify and counter the TLB filtering effect is warranted.*

Aborted and wrong path walks are significant. Aborted and wrong path page table walks constitute a significant fraction of all page table walks, which gets worse with increasing memory footprints. This constitutes a significant cost in terms of energy, cache bandwidth, and cache footprint - especially at larger memory footprints where the TLB miss rate is high. *We believe that further investigation into the causes of aborted and wrong path walks is required.*

Superpages do not solve everything Although superpages do lead to large reductions in address translation pressure, they do not make it go away completely - especially for larger memory footprints. As footprints continue to grow, it appears likely that we will encounter the same problems that we currently seeing with 4KB pages. *Practioners should investigate the effectiveness of their optimizations with superpages enabled - especially for terabyte-scale applications.*

We analyzed sources of error through comparison against our CounterPoint-derived Haswell MMU model. We find the presence of undocumented microarchitectural features such as *TLB miss merging* may introduce errors in how walk overhead is attributed to individual sources. Overall, the WCPI metric remains valid, and accurately predicts address translation overhead.

Chapter 4

Conclusions

This thesis advances hardware event counter driven microarchitectural research and our understanding of address translation performance and implementation on real hardware.

Model constraints. Researchers use hardware event counters to investigate microarchitectural behavior [2, 3, 157, 85]. This thesis introduces the notion of a *model constraint* - an equality or inequality defined over event counter values that enable experts to test their microarchitectural assumptions. This thesis demonstrates that model constraints are powerful and can be used to infer fine-grained microarchitectural features. This thesis empirically demonstrates why manually deriving and evaluating model constraints does not scale with the increasing number of hardware event counters available on commodity hardware, motivating an automated approach.

The CounterPoint approach. This thesis introduces CounterPoint, a methodology and automated tool that enables experts to build microarchitectural models that explain hardware event counter observations. The key insight is that testing HEC measurements for consistency with model constraints can be precisely formulated as a linear programming problem. The program is derived directly from a microarchitecture model and event counter measurement, and can be solved efficiently by modern solvers, thereby overcoming the limitations associated with manually deriving and evaluating model constraints. CounterPoint uses statistical confidence regions to mitigate the error introduced by time-multiplexing hardware event counters.

Evaluating CounterPoint with a case study. This thesis demonstrates CounterPoint is fast and effective with a case study on the memory management unit implementation on the Intel Haswell microarchitecture. This case study presents evidence for a number of sophisticated microarchitectural behaviors, including: TLB miss merging, early paging structure cache lookup,

and prefetcher-triggered page table walks that abort when unset accessed bits are encountered.

Situating CounterPoint in the microarchitectural research space. CounterPoint has the potential to impact the broader space of microarchitectural research using hardware event counters. To illustrate, this thesis revisits a research study we performed on memory management unit performance under workloads with increasing memory footprint. We found that: address-translation performance overhead that scales with the log of the memory footprint; that large pages cannot hide this behavior; and that a significant portion of aborted and misspeculated page table walks. This peer-reviewed study was published prior to CounterPoint's invention. Comparing the event counter metrics used by the study against the CounterPoint-derived MMU model suggests that the high level conclusions remain valid, but that there *may* be some errors in how overhead is attributed to individual MMU components. This demonstrates how CounterPoint can be a useful tool for evaluating the strengths and weaknesses of HEC driven research studies.

4.0.1 Future research directions

This thesis opens up several promising avenues for future research:

Accelerators and system-wide modeling. Most advanced computing systems today are heterogeneous, with accelerators such as GPUs [103, 42, 102], TPUs [72, 71, 105] and SmartNICs [79, 145], integrated through system-wide interconnects such as PCIe. Many of these components are instrumented with hardware event counters [80, 158, 72]. CounterPoint could improve the research community's understanding of these devices, enabling more fine-grained performance insights and the construction of more detailed software simulators.

A key assumption behind CounterPoint is that every event captured by hardware event counters can be attributed to some common object (like a micro-op or instruction). For accelerators with different execution models (such as GPUs and TPUs), such a clear assignment may not exist. The events will remain related to each other in specific ways, but new abstractions may need to be developed to encode this in a mathematically precise way that allows for automated testing.

Duration-style event counters. Beyond discrete events (*e.g.* cache hits/misses), modern hardware event counters can also record the duration of microarchitecture conditions (*e.g.* processor stalls due to outstanding memory accesses). Understanding duration style events is important

because they can be closely correlated with overall runtime (as we showed for page table walk duration cycles in Chapter 3), thus making them useful tools in predicting performance. Knowledge of the duration of microarchitectural events would also reveal more insight into the microarchitectural implementation, allowing for the better calibration of microarchitectural simulators.

To benefit from duration-style event counters, CounterPoint could be extended to support an explicit notion of time. This would require enabling the μpath Decision Diagram model representation to (i) support duration style event counters and (ii) an ability to associate microarchitectural events with latencies. Updating the linear program formulation of model consistency to support latencies is an interesting research challenge: it is unclear if the model cone abstraction would work with latencies in its current form, and a more sophisticated construction may be required.

Automatically mapping performance metrics to event counters. Today, there are no publicly known methods for automatically mapping performance metrics (like cache hit rates) to explicit formulae over event counter values. We showed in Chapter 3 that manually constructing event counter formulae can inadvertently introduce systematic error - even when the full microarchitectural model is assumed to be known. This may be because the mapping is sub-optimal (*e.g.* there is another mapping that more accurately captures the desired quantity), or because there does not exist any mapping (*e.g.* no combination of event counters have the required semantics).

Given a microarchitectural model, it may be possible to automatically map a machine-readable performance metric specification to an explicit event counter formula. This would bring several benefits. First, it would allow performance metrics to be quickly ported to a new microarchitecture by simply updating the microarchitectural model. Evaluating metrics at different stages through the CounterPoint model refinement process would also be helpful. Second, evaluating whether or not a metric can be mapped to event counters would help vendors evaluate the tradeoff between providing valuable performance debugging information versus the ease by which microarchitectural trade secrets can be inferred from their event counters. Third, a precise specification language for performance metrics would facilitate reproducibility across research studies.

4.0.2 Closing remarks

This thesis demonstrates that new abstractions, rooted in formal methods and statistics, can provide a solid foundation for bridging the gap between the opportunities and limitations of hardware event counters.

Bibliography

[1] 7CPU Website Author(s). Intel haswell (7cpu). `https://www.7-cpu.com/cpu/Haswell.html`. [Accessed 02-06-2024].

[2] Andreas Abel and Jan Reineke. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, 2019.

[3] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46. IEEE, 2020.

[4] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[5] Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. Predicting execution times with partial simulations in virtual memory research: Why and how. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 2020-Octob, pages 456–470. IEEE Computer Society, oct 2020.

[6] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[7] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. *ACM SIGARCH Computer Architecture News*, 40(3):476–487, 2012.

[8] Sam Ainsworth and Timothy M. Jones. Compendia: Reducing virtual-memory costs via selective densification. *ISMM 2021 - Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management, co-located with PLDI 2021*, pages 52–65, 2021.

[9] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 457–468, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th*

Annual International Conference on Supercomputing, ICS '05, page 101–110, New York, NY, USA, 2005. Association for Computing Machinery.

[11] Reza Azimi, David K Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, 2009.

[12] Vlastimil Babka and Petr Tuma. Investigating cache parameters of x86 family processors. In *SPEC Benchmark Workshop*, pages 77–96. Springer, 2009.

[13] Subho S. Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. BayesPerf: Minimizing performance monitoring errors using Bayesian statistics. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 832–844, 2021.

[14] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *SIGARCH Comput. Archit. News*, 38(3):48–59, jun 2010.

[16] Daniel Barry, Anthony Danalis, and Jack Dongarra. Automated data analysis for defining performance metrics from raw hardware events. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 716–725. IEEE, 2024.

[17] Daniel Barry, Anthony Danalis, and Heike Jagode. Effortless monitoring of arithmetic intensity with papi's counter analysis toolkit. In *Tools for High Performance Computing 2018/2019: Proceedings of the 12th and of the 13th International Workshop on Parallel Tools for High Performance Computing, Stuttgart, Germany, September 2018, and Dresden, Germany, September 2019*, pages 195–218. Springer, 2021.

[18] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.

[19] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[20] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[21] Andrin Bertschi. Battling the prefetcher: Exploring coffee lake (part 1). `https://abertschi.ch/blog/2022/prefetching/#intel-manuals-on-prefetching`, 2022. Accessed: 2025-04-29.

[22] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[23] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. *VMware, Inc*, 2009.

[24] Abhishek Bhattacharjee. Large-reach memory management unit caches. *MICRO 2013 - Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 383–394, 2013.

[25] Abhishek Bhattacharjee. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro*, 37(5):6–10, 2017.

[26] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–76, 2017.

[27] Abhishek Bhattacharjee. Appendix l: Advanced concepts on address translation, 2019.

[28] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 62–73, 2011.

[29] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. *Architectural and operating system support for virtual memory*. Springer, 2018.

[30] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. *ACM Sigplan Notices*, 45(3):359–370, 2010.

[31] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[32] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 72–81, 2008.

[33] William Lloyd Bircher and Lizy K. John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2012.

[34] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. Improving virtualization in the presence of software managed translation lookaside buffers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 120–129, 2013.

[35] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A. Jiménez, and Marc Casas. Instruction-aware cooperative tlb and cache replacement policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 619–636, New York, NY, USA, 2025. Association for Computing Machinery.

[36] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 293–304, 2012.

[37] COIN-OR Project. coin-or/cbc: Release releases/2.10.12. *Zenodo*, August 2024.

[38] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[39] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[40] Peter Cordes. What happens after a l2 tlb miss? `https://stackoverflow.com/a/32258855`, 2021. Accessed: April 6th, 2025.

[41] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. *ACM SIGPLAN Notices*, 52(4):435–448, 2017.

[42] William J Dally, Stephen W Keckler, and David B Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

[43] Anthony Danalis, Heike Jagode, Hanumantharayappa, Sangamesh Ragate, and Jack Dongarra. Counter inspection toolkit: Making sense out of hardware performance events. In *International Workshop on Parallel Tools for High Performance Computing*, pages 17–37. Springer, 2017.

[44] Nirav Dave, Man Cheuk Ng, et al. Automatic synthesis of cache-coherence protocol processors using bluespec. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.*, pages 25–34. IEEE, 2005.

[45] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH computer architecture news*, 41(3):559–570, 2013.

[46] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[47] Gem5 Discussion. Riscv Virtual Address Translation Process. *https://github.com/orgs/gem5/discussions/1220*, 2024.

[48] Gem5 Discussion. Adding realistic SMMU invalidation delays to gem5. *https://github.com/orgs/gem5/discussions/2227*, 2025.

[49] Joel S Emer and Douglas W Clark. A characterization of processor performance in the vax-11/780. *ACM SIGARCH Computer Architecture News*, 12(3):301–310, 1984.

[50] Philipp Ertmer, Robert Dumitru, and Yuval Yarom. Reverse-engineering the address translation caches. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 149–168. Springer, 2025.

[51] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A top-down approach to architecting cpi component performance counters. *IEEE micro*, 27(1):84–93, 2007.

[52] Vadim Filanovsky and Harshad Sane. Seeing through hardware counters: a journey to threefold performance increase. *Netflix Blog*, 2022.

[53] Komei Fukuda. What is the minkowski–weyl theorem for convex polyhedra? Polyhedral Computation FAQ, August 2004. Accessed via ETH Zurich website (people.inf.ethz.ch/fukudak/polyfaq/).

[54] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile paging: Exceeding the best of nested and shadow paging. *ACM SIGARCH Computer Architecture News*, 44(3):707–718, 2016.

[55] Andy Glew, Glenn Hinton, and Haitham Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions, October 21 1997. US Patent 5,680,565.

[56] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. Mosaic Pages: Big TLB Reach with Small Pages. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, volume 3, pages 433–448. Association for Computing Machinery, 3 2023.

[57] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *ACM SIGMOD Record*, 23(2):243–252, 1994.

[58] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with Midgard. *Proceedings - International Symposium on Computer Architecture*, 2021-June:512–525, 2021.

[59] Faruk Guvenilir and Yale N. Patt. Tailored Page Sizes. *Proceedings - International Symposium on Computer Architecture*, 2020-May:900–912, 2020.

[60] Nathaniel E. Helwig. Inferences about multivariate means. `http://users.stat.umn.edu/~helwig/notes/mvmean-Notes.pdf`, 2017. Lecture notes, University of Minnesota.

[61] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[62] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[63] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests. *Proceedings - International Symposium on Computer Architecture*, 2020-May:874–887, 2020.

[64] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 679–694, 2021.

[65] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–524. IEEE, 2024.

[66] Intel Corporation. Intel Performance Monitoring Events (perfmon). `https://github.com/intel/perfmon/tree/main`.

[67] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes (PDF). `https://cdrdv2.intel.com/v1/dl/getContent/671200`, October 2025. Published October 2025.

[68] International Business Machines Corporation. *POWER9 Performance Monitor Unit User's Guide*. IBM Systems, Somers, NY, USA, version 1.2 edition, November 2018.

[69] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 93–104. IEEE, 2003.

[70] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely, and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal Locality Aware (TLA) cache management policies. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 151–162, 2010.

[71] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[72] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann,

C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Tomasz Jurkiewicz and Kurt Mehlhorn. On a model of virtual address translation. *ACM Journal of Experimental Algorithmics*, 19(1):1–29, 2015.

[74] Gokul B Kandiraju and Anand Sivasubramaniam. Going the distance for tlb prefetching: An application-driven study. *ACM SIGARCH Computer Architecture News*, 30(2):195–206, 2002.

[75] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1178–1195, 2023.

[76] Konstantinos Kanellopoulos, Konstantinos Sgouras, F. Nisa Bostanci, Andreas Kosmas Kakolyris, Berkin Kerim Konar, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Nandita Vijaykumar, and Onur Mutlu. *Virtuoso: Enabling Fast and Accurate Virtual Memory Research via an Imitation-based Operating System Simulation Methodology*. Association for Computing Machinery, New York, NY, USA, 2025.

[77] Konstantinos Kanellopoulos, Konstantinos Sgouras, F. Nisa Bostanci, Andreas Kosmas Kakolyris, Berkin Kerim Konar, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Nandita Vijaykumar, and Onur Mutlu. Virtuoso: Enabling fast and accurate virtual memory research via an imitation-based operating system simulation methodology, 2025.

[78] Shinichi Kawaguchi and Toshiaki Yachi. Adaptive power efficiency control by computer power consumption prediction using performance counters. *IEEE Transactions on Industry Applications*, 52(1):407–413, 2016.

[79] Elie F Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access*, 12:107297–107336, 2024.

[80] Hyesoon Kim, Richard Vuduc, and Sara Baghsorkhi. *Performance analysis and tuning for general purpose graphics processing units (GPGPU)*, volume 20. Morgan & Claypool Publishers, 2012.

[81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201, 1998.

[82] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N Patt. Improving memory bank-level parallelism in the presence of prefetching. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–336, 2009.

[83] Hyun Ryong Lee and Daniel Sanchez. Datamime: Generating Representative Benchmarks by Automatically Synthesizing Datasets. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2022-Octob:1144–1159, 2022.

[84] Kyeong-Jae Lee and Kevin Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12*, IPDPS '05, page 232.1, USA, 2005. IEEE Computer Society.

[85] Nick Lindsay and Abhishek Bhattacharjee. Understanding address translation scaling behaviours using hardware performance counters. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pages 236–246. IEEE, 2024.

[86] Nick Lindsay, Caroline Trippel, Anurag Khandelwal, and Abhishek Bhattacharjee. Counterpoint: Using hardware event counters to refute and refine microarchitectural assumptions (extended version), 2026.

[87] Jason Lowe-Power. PIPT/VIPT caches and TLB lookup latency. *Gem5 Discussion*, page https://github.com/orgs/gem5/discussions/2227, 2025.

[88] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646. IEEE, 2014.

[89] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipe Check: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2015-Janua(January):635–646, 2015.

[90] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. *ACM SIGPLAN Notices*, 51(4):233–247, apr 2016.

[91] Yirong Lv, Bin Sun, Qingyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. Counterminer: Mining big performance data from hardware counters. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:613–626, 2018.

[92] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:788–801, 2018.

[93] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using νhb graphs to verify the coherence-consistency interface. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 05-09-Dece:26–37, 2015.

[94] Aninda Manocha, Zi Yan, Esin Tureci, Juan Luis Aragón, David Nellans, and Margaret Martonosi. The implications of page size management on graph analytics. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 199–214. IEEE, 2022.

[95] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036, 2019.

[96] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 1023–1036, 2019.

[97] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. Ptemagnet: Fine-grained physical memory reservation for faster page walks in public clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, 2021.

[98] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[99] Timothy Merrifield and H Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 25–35, 2016.

[100] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. BDGS: A scalable big data generator suite in big data benchmarking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8585:138–154, 2014.

[101] Stuart Mitchell, Michael OSullivan, and Iain Dunning. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 65:25, 2011.

[102] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015.

[103] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[104] Chase Norman, Adwait Godbole, and Yatin A Manerkar. Pipesynth: automated synthesis of microarchitectural axioms for memory consistency. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 513–527, 2023.

[105] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The Design Process for Google's Training Chips: TPUv2 and TPUv3 . *IEEE Micro*, 41(02):56–63, March 2021.

[106] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 247–260. IEEE, 2018.

[107] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 888–899. IEEE, 2020.

[108] Nicolai Oswald, Vijay Nagarajan, Daniel J Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. Heterogen: Automatic synthesis of heterogeneous cache coherence protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 756–771. IEEE, 2022.

[109] Yueyang Pan, Yash Lala, Musa Unal, Yujie Ren, Seung seob Lee, Yizhou Shan, Abhishek Bhattacharjee, Anurag Khandelwal, and Sanidhya Kashyap. Scalable Far Memory: Balancing Faults and Evictions. *SOSP*, 2024.

[110] The pandas development team. pandas-dev/pandas: Pandas. *Zenodo*, July 2025.

[111] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021.

[112] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[113] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Efficient synonym filtering and scalable delayed translation for hybrid virtual caching. *ACM SIGARCH Computer Architecture News*, 44(3):217–229, 2016.

[114] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every walk'sa hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–141, 2022.

[115] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 558–567, 2014.

[116] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 258–269, USA, 2012. IEEE Computer Society.

[117] Binh Pham, Ján Veselỳ, Gabriel H Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 1–12, 2015.

[118] Fong Pong and Michel Dubois. The verification of cache coherence protocols. In *Proceedings of the fifth annual ACM symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[119] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[120] Hongliang Qu and Zhibin Yu. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2:1233–1249, 2024.

[121] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. A data generator for cloud-scale benchmarking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6417 LNCS:41–56, 2011.

[122] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. *Proceedings of the 30th USENIX Security Symposium*, pages 1451–1468, 2021.

[123] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 1106–1120, 2021.

[124] Benny Rubin, Saksham Agarwal, Qizhe Cai, and Rachit Agarwal. Fast & Safe IO Memory Protection. *SOSP*, 2024.

[125] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB . In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 469–480, Los Alamitos, CA, USA, June 2017. IEEE Computer Society.

[126] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[127] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based tlb preloading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 117–127, 2000.

[128] Balvinder Pal Singh and B Thangaraju. Power, performance and thermal management using hardware performance counters. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–7, 2020.

[129] Richard L Sites. Performance counters i'd like to see – part ii. `https://www.sigarch.org/performance-counters-id-like-to-see-part-ii/`, jan 2022. [Accessed 05-01-2025].

[130] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1093–1108, 2020.

[131] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 1093–1108, 2020.

[132] Standard Performance Evaluation Corporation. Spec cpu2006. `https://www.spec.org/cpu2006/`. [Accessed 02-06-2024].

[133] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. *ISCA*, 2013.

[134] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. *HPCA*, 2013.

[135] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. {TLB; DR}: Enhancing {TLB-based} attacks with {TLB} desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 989–1007, 2022.

[136] The Linux Kernel Developers. pmu-events: Performance Monitoring Unit (PMU) events definitions in Linux perf. `https://github.com/torvalds/linux/tree/master/tools/perf/pmu-events`.

[137] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:947–960, 2018.

[138] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 409–422. IEEE, 2006.

[139] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A framework for reverse engineering hardware page table caches. *Proceedings of the Proceedings of the 10th European Workshop on Systems Security, EuroSec 2017, co-located with European Conference on Computer Systems, EuroSys 2017*, 2017.

[140] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Reverse engineering hardware page table caches using side-channel attacks on the mmu. *Vrije Universiteit Amsterdam, Tech. Rep*, 2017.

[141] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. Morrigan: A composite instruction tlb prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1138–1153, 2021.

[142] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A Jiménez, and Marc Casas. Exploiting page table locality for agile tlb prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 85–98. IEEE, 2021.

[143] Alan Wang, Boru Chen, Yingchen Wang, Christopher Fletcher, Daniel Genkin, David Kohlbrenner, and Riccardo Paccagnella. Peek-a-walk: Leaking secrets via page walk side channels. In *2025 IEEE Symposium on Security and Privacy (S&P). IEEE.[Online]. Available: https://www. computer. org/csdl/proceedings-article/sp/2025/22360 0a023/21B7QepK7Fm*, 2024.

[144] Xueyang Wang and Ramesh Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th annual design automation conference*, pages 1–7, 2013.

[145] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path {SmartNIC} for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, 2023.

[146] Dee A.B. Weikle, Sally A. McKee, and Wm A. Wulf. Caches as filters: A new approach to cache analysis. *IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems - Proceedings*, pages 2–12, 1998.

[147] Henry Wong. TLB and pagewalk coherence in x86 processors, aug 2015.

[148] Yu Xia, Vishnu Ramadas, Matthew Poremba, and Matthew D. Sinclair. Narrowing the GAP: Enhancing gem5's GPU Memory Bandwidth Accuracy. *Gem5 Workshop*, 2025.

[149] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware TLBs. *Proceedings - International Symposium on Computer Architecture*, pages 698–710, 2019.

[150] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 430–443, 2017.

[151] Xi Yang, Stephen Blackburn, and Kathryn McKinley. Computer Performance Microscopy with SHIM. *ISCA*, 2015.

[152] Yihao Yang, Pengfei Qiu, Chunlu Wang, Yu Jin, Qiang Gao, Xiaoyong Li, DongSheng Wang, and Gang Qu. Exploration and exploitation of hidden pmu events. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[153] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 35–44, 2014.

[154] Reza Zamani and Ahmad Afsahi. A study of hardware performance monitoring counter selection in power modeling of computing systems. In *2012 International Green Computing Conference (IGCC)*, pages 1–10. IEEE, 2012.

[155] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct Memory Translation for Virtualized Clouds. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2:287–304, 2024.

[156] Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen. Processor hardware counter statistics as a first-class system resource. In *HotOS*, 2007.

[157] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker. *Proceedings of the 31st USENIX Security Symposium, Security 2022*, pages 699–716, 2022.

[158] Bob Zigon and Fengguang Song. Utilizing gpu performance counters to characterize gpu kernels via machine learning. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 88–101, Cham, 2020. Springer International Publishing.

Appendix A

Linear Program Formulation

We construct and solve the following linear program to determine the feasibility of microarchitectural observations against a μDD model:

$$\left[\begin{array}{l} \vec{v} \in \mathbb{R}_+^N \text{ (Counter variables)} \\ \forall p \in \mathcal{P}(D) \;.\; f(p) \in \mathbb{R}_+ \text{ (Flow variables)} \\ \vec{v} = \sum_{p \in \mathcal{P}(D)} \vec{S}(p) \cdot f(p) \text{ (Counter flow equation)} \\ \forall i \in n \;.\; |\vec{e_i} \cdot (\vec{v} - \bar{Y})| \leq \sqrt{\lambda_i \chi^2_{d,1-\alpha}} \\ \text{(Counter confidence region encoding)} \end{array}\right] \tag{LP}$$

Each path through the μDD is enumerated by breadth-first search. Variables are instantiated for the the true counter values $\vec{v}$ and the flow $f(p)$ down each μpath. The variables are constrained to be non-negative. Counter and flow variables are related by the Counter Flow Equation, which implicitly describes the model cone.

The confidence ellipsoid cannot be directly encoded in the linear program as it is a quadratic form. Instead, the bounding box is constructed - aligning edges to the principle axes of the ellipsoid (Figure 2.4b). The directions of the principle axes of the confidence ellipsoid are determined by the normalized eigenvectors $\vec{e_1}, ..., \vec{e_n}$ of the covariance matrix. The half-length of the ith axis is given by $\sqrt{\lambda_i \chi^2_{N,1-\alpha}}$, where λ_i is the ith eigenvalue. Figure 2.4b graphically depicts the construction for systems with two counters.