Understanding Address Translation Scaling Behaviours Using Hardware Performance Counters

Nick Lindsay
Department of Computer Science
Yale University
New Haven, USA
nick.lindsay@yale.edu

Abhishek Bhattacharjee
Department of Computer Science
Yale University
New Haven, USA
abhishek@cs.yale.edu

***Abstract*—Virtual memory researchers are combining benchmark programs with synthetic input generators when crafting experiments. These generators are tuned to generate program instances with memory footprints in the gigabyte to terabyte range, with the intention that these instances will have considerable address translation overhead. Yet, the relationship between workload, input, and address translation overhead is poorly understood - complicating workload and input selection.**

We characterize this relationship on real machines across various workloads, input generators, and memory footprints using measured performance counter data. We observe that address translation overhead typically increases as the $\log$ of the memory footprint, but there are exceptions - even for workloads that are thought of as being "address translation intensive".

We measure and provide results for the memory footprint scaling behavior of the program, translation lookaside buffers (TLBs), memory management unit (MMUs) caches, and page table walkers. We show that no individual factor is responsible for degrading performance but rather the combination of all four (and their interactions) gives rise to address translation-related slowdown. We find evidence for the existence of a TLB filtering effect: higher TLB hit rates can cause longer page table walks because the TLB filters page-level access patterns from the MMU caches and page table walker(s).

We propose a new metric of address translation pressure called "walk cycles per instruction" that captures the effects of all the components and interactions. We demonstrate that it strongly reflects true address translation overhead.

Finally, we show that misspeculated and aborted page table walks can constitute up to 57% of all initiated page table walks and that the problem gets worse with increasing memory footprint. Superpages can reduce the number of misspeculated and aborted walks, suggesting that superpages have effects beyond reducing TLB miss rate and shortening page table walks.

***Index Terms*—address translation, virtual memory, workload characterization, performance counters**

I. 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 [8]. As a result numerous optimizations have been proposed [4], [7], [10], [14], [16], [17], [21], [24], [26], [27], [29], [30], [32], [34], [35].

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 [3], [6], [11]. 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 [17] and the ease with which the corresponding memory footprint can be accommodated in the simulator [22]. However, the precise relationship between the input size and these various factors is not well understood. Understanding this is important for simplifying 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.

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?

The authors acknowledge the support of the National Science Foundation grant number 2112562, Intel, and Meta.

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

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

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. *§V-A*.
- Walk cycles per instruction is a good proxy for AT overhead. *§V-B*.
- 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. *§V-C*.
- Wrong path and aborted page table walks constitute a significant fraction of all walks. *§V-D*.
- The benefits of 2MB pages start to expire for workloads with footprints over 100GBs - within the range of footprints for modern-day workloads. *§V-E*.

II. Background and Prior Work

A. Address translation

Address translation is the process of converting application-visible **virtual addresses** to hardware **physical addresses** [9], [18]. 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 so expensive, processors include specialized caches to avoid them. **Translation lookaside buffers** (TLBs) cache the results of the most recent page table walks, 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** [2]. Paging structure caches cache partial page table walks [5], 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.

B. Workload selection

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 [3] and PARSEC [11]. 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 [7], [14], [16], [24], [26], [27], [34], [35] .

There has been work on creating input generators that can replicate the characteristics of known but proprietary inputs [15], [23], [25], [28]. Our work is orthogonal to these approaches and could be combined to create larger workload instances with similar characteristics.

III. Quantifying overhead

In this section we propose a definition for "address translation (AT) overhead" and discuss how to estimate it experimentally (Section III-A). We justify our choice of baseline (Section III-B). We introduce Walk Cycles per Instruction as a proxy for AT overhead (SectionIII-C).

A. 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*

[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.

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:

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

B. Explanation of baseline

We use $\min(t_{\text{2MB}}, t_{\text{1GB}})$ 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.

C. 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 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 V-B.

TABLE I
WORKLOADS
(ST = SINGLE-THREADED, MT = MULTITHREADED)

Suite	Program	Generators	Type
gapbs [6]	bc, bfs, cc, pr, tc	urand, kron	graph processing (MT)
ycsb [13]	memcached	uniform	key-value store (MT)
spec2006 [3]	mcf	rand	network simplex (ST)
parsec [11]	streamcluster	rand	clustering (MT)

TABLE II
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 $N(N-1)$ dead-head trips
(streamcluster) rand	list of random vectors

IV. METHODOLOGY

We select a range of programs that are typically considered address translation intensive (Table I). These include graph processing, network simplex, key-value stores, and clustering algorithms. We use the corresponding input generators listed in Table II. `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 corresponding memory footprint in the 4KB configuration since this is a more tangible quantity.

Table III 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);

TABLE III
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

and warming up file system caches with a 60 second dry run of the experiment.

V. Results and Analysis

A. 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 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 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 [20] 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 IV.

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.

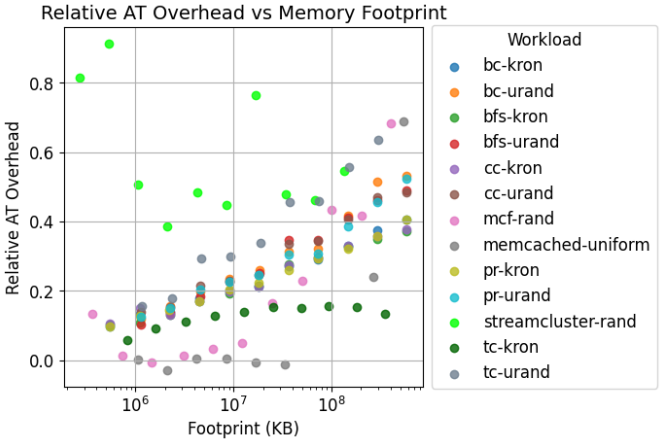

Fig. 1. Relationship between relative AT overhead and memory footprint, grouped by workload.

There are four exceptions to this trend: `mcf-rand`, `memcached-uniform`, `streamcluster-rand`, and `tc-kron`. We plot their trends in Figure 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 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`) [6]. 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".

B. 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 transla-

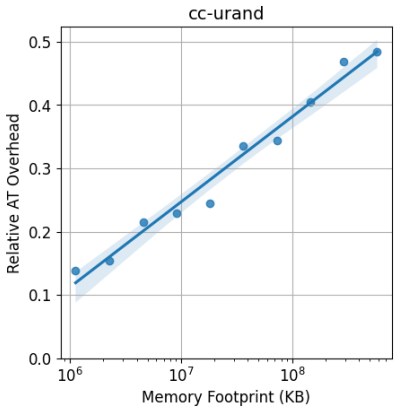

Fig. 2. Relative AT overhead vs memory footprint for `cc-urand`

TABLE IV
REGRESSION RESULTS FOR MODEL
RELATIVE AT OVERHEAD $= \beta_0 + \beta_1 \log_{10} M + \epsilon$.

Program	Generator	Coefficients const	Coefficients $\log_{10} M$	Statistic Adj. R^2
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

tion 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

TABLE V
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

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 V shows the various correlation coefficients between AT pressure 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 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 single workload. We select `bc-urand` as a representative workload and plot the overhead-WCPI relationship in Figure 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

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

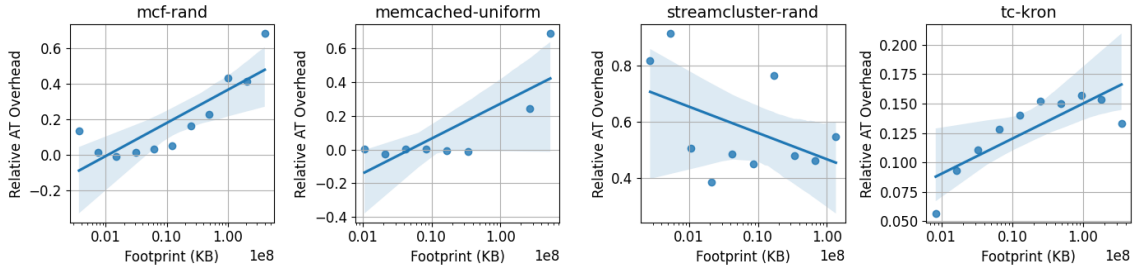

Fig. 3. Relative AT overhead vs memory footprint for workloads with weaker linear correlations.

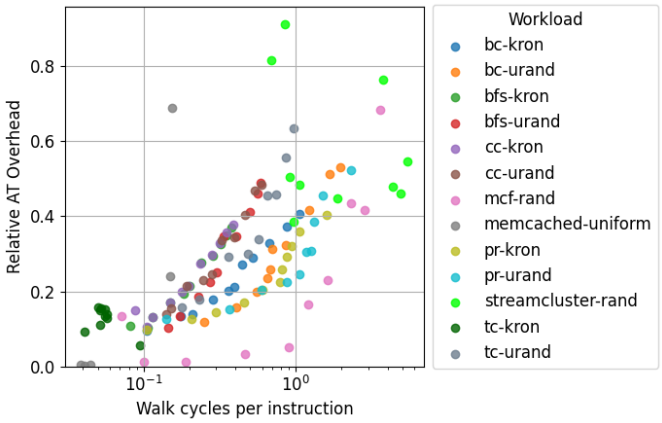

Fig. 4. Relationship between relative AT overhead and walk cycles per instruction, grouped by workload. Workload-input size combinations with a negative measured relative AT overhead are assumed to not be AT sensitive and are excluded from the figure.

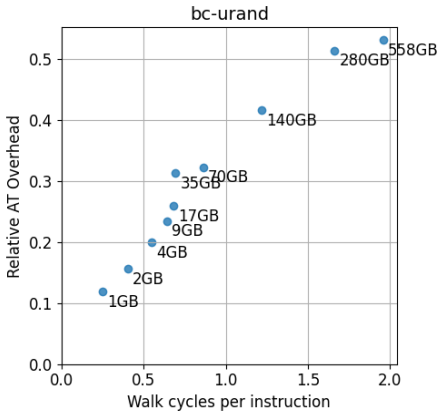

Fig. 5. Relationship between AT overhead and WCPI for `bc-urand`. Each point is labeled by memory footprint.

change as the input size varies; for example, see the second row of graphs in Figure 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.

C. 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 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.

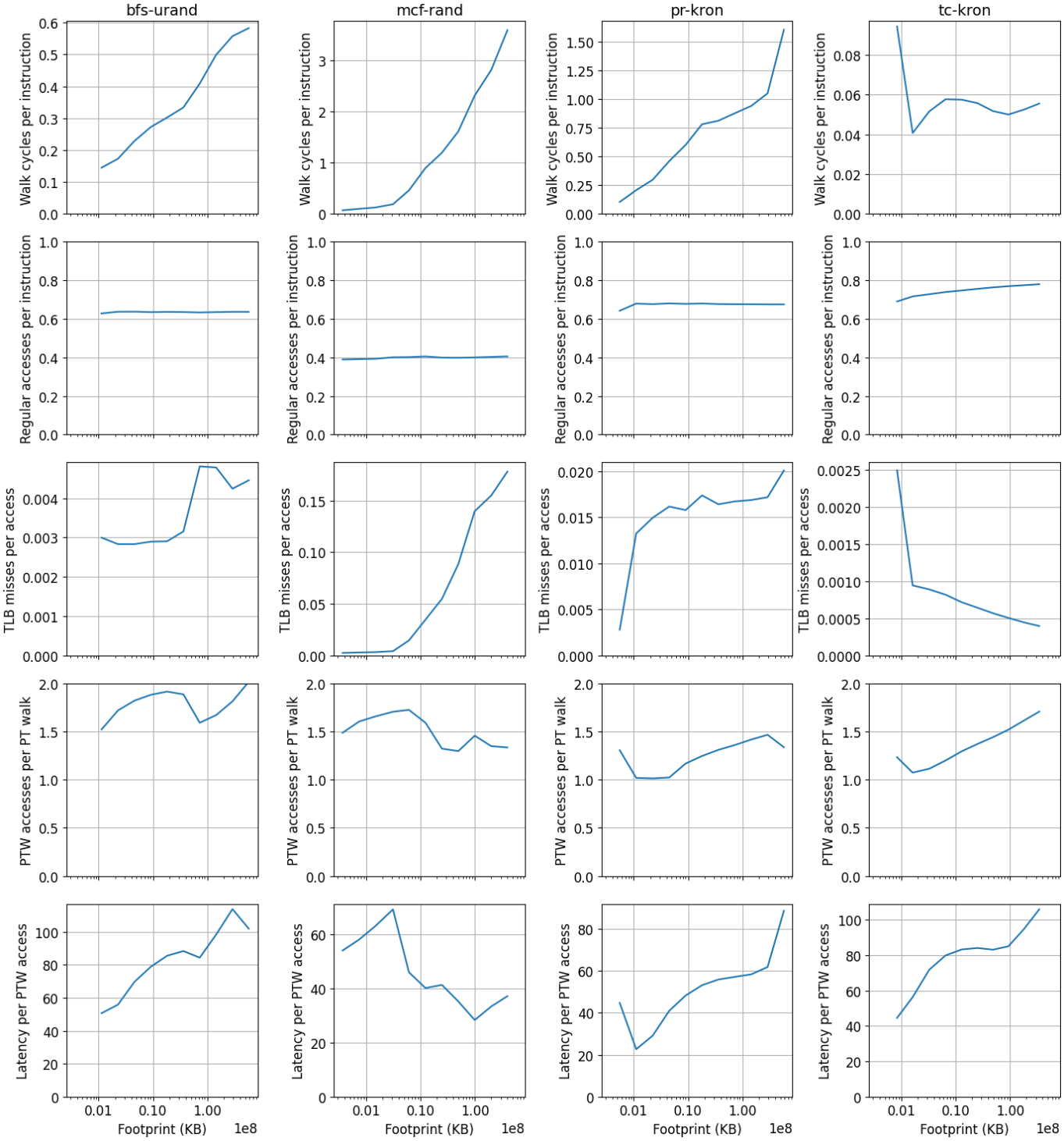

Fig. 6. Component-wise breakdown of the scaling behavior of four workloads.

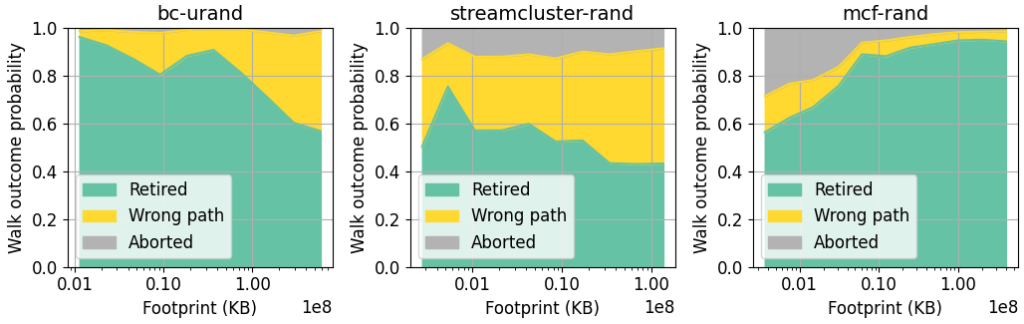

Fig. 7. Walk outcome distribution as a function of memory footprint. The vertical width of each band corresponds to the probability that any initiated walk has that outcome.

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 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 V-D.

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 caches [31], 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 [12], [19], [33]. 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.

TABLE VI
WALK OUTCOME METRIC FORMULAE

Walk outcome	Formula
Initiated	`dtlb_load_misses.miss_causes_a_walk` + `dtlb_store_misses.miss_causes_a_walk`
Completed	`dtlb_load_misses.walk_completed` + `dtlb_store_misses.walk_completed`
Retired	`mem_uops_retired.stlb_miss_loads` + `mem_uops_retired.stlb_miss_stores`
Aborted	Initiated - Completed
Wrong path	Completed - Retired

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 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 average latency

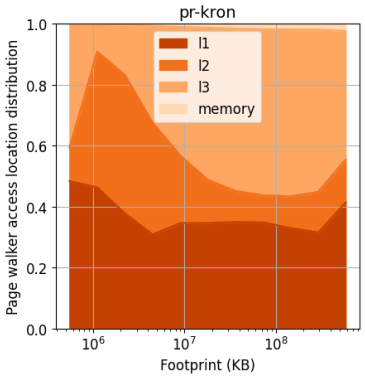

Fig. 8. Distribution of PTE access location as a function of input size for `pr-kron`. The vertical width of each band is the probability that the page table walker finds the PTE in that location.

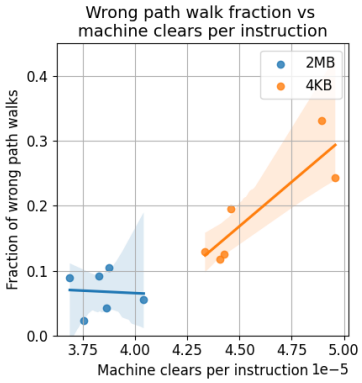

Fig. 9. Wrong path walk fraction vs machine clears per instruction.

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.

D. 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** before they are finished. Table VI shows how we compute counts from performance counters.

We plot the outcomes of page table walks for 3 workloads in Figure 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 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.

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.

E. 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 [29], 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 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

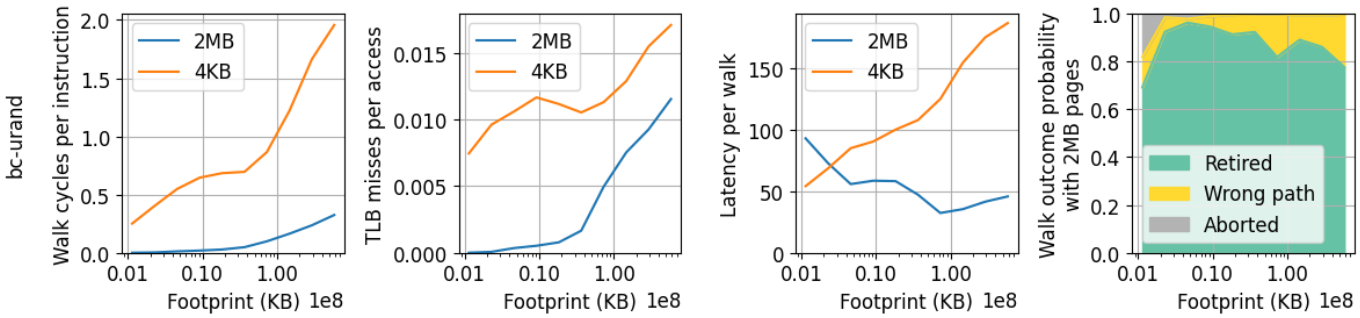

Fig. 10. Key address translation metrics for `bc-urand` with 2MB pages. Comparison with 4KB pages included for all graphs except walk outcomes.

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 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.

VI. Discussion

In this work we set out to understand the complex interplay between program, input, address translation pressure, and address translation overhead. 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 [12], [19], [33]. *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.*

Acknowledgment

The authors acknowledge the support of the National Science Foundation grant number 2112562, Intel, and Meta. We appreciate the anonymous reviewers and shepard for their comments and guidance on our paper. We thank Andrew Milas for his insightful observations.

References

[1] "Intel haswell (7cpu)," https://www.7-cpu.com/cpu/Haswell.html, [Accessed 02-06-2024].

[2] "Intel® 64 and IA-32 Architectures Software Developer Manuals — intel.com," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, [Accessed 03-06-2024].

[3] "Spec cpu2006," https://www.spec.org/cpu2006/, [Accessed 02-06-2024].

[4] S. Ainsworth and T. 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*, pp. 52–65, 2021.

[5] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 48–59, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815970

[6] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," pp. 1–16, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[7] A. Bhattacharjee, "Translation-Triggered Prefetching."

[8] ——, "Expert Opinion The Challenges Facing Virtual Memory Today," 2017. [Online]. Available: www.computer.org/micro

[9] A. Bhattacharjee and D. Lustig, "Architectural and Operating System Support for Virtual Memory," *Synthesis Lectures on Computer Architecture*, pp. 1–156, 2018.

[10] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 62–73, 2011.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 72–81, 2008.

[12] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches," *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 293–304, 2012.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," pp. 143–154.

[14] K. Gosakan, W. Kuszmaul, I. N. Mubarek, G. Tagliavini, E. West, M. A. Bender, A. Conway, M. Farach-colton, R. Johnson, and D. E. Porter, "Mosaic Pages : Big TLB Reach with Small Pages," pp. 433–448.

[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly Generating Billion-Record Synthetic Databases," *ACM SIGMOD Record*, vol. 23, no. 2, pp. 243–252, 1994.

[16] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting virtual memory with Midgard," *Proceedings - International Symposium on Computer Architecture*, vol. 2021-June, pp. 512–525, 2021.

[17] F. Guvenilir and Y. N. Patt, "Tailored Page Sizes," *Proceedings - International Symposium on Computer Architecture*, vol. 2020-May, pp. 900–912, 2020.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[19] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely, and J. 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*, pp. 151–162, 2010.

[20] T. Jurkiewicz and K. Mehlhorn, "On a model of virtual address translation," *ACM Journal of Experimental Algorithmics*, vol. 19, no. 1, pp. 1–29, 2015.

[21] K. Kanellopoulos, H. C. Nam, F. N. Bostanci, R. Bera, M. Sadrosadati, R. Kumar, D.-B. Bartolini, and O. Mutlu, "Victima: Drastically Increasing Address Translation Reach by Leveraging Underutilized Cache Resources," pp. 1–18, 2023. [Online]. Available: http://arxiv.org/abs/2310.04158%0Ahttp://dx.doi.org/10.1145/3613424.3614276

[22] K. Kanellopoulos, K. Sgouras, and O. Mutlu, "Virtuoso: An Open-Source, Comprehensive and Modular Simulation Framework for Virtual Memory Research," 2024. [Online]. Available: http://arxiv.org/abs/2403.04635

[23] H. R. Lee and D. Sanchez, "Datamime: Generating Representative Benchmarks by Automatically Synthesizing Datasets," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2022-Octob, pp. 1144–1159, 2022.

[24] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 1023–1036, 2019.

[25] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. 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)*, vol. 8585, pp. 138–154, 2014.

[26] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: making page walks single-access cache hits," pp. 128–141, 2022.

[27] H. Qu and Z. Yu, "WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 2, pp. 1233–1249, 2024.

[28] T. Rabl, M. Frank, H. M. Sergieh, and H. 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)*, vol. 6417 LNCS, pp. 41–56, 2011.

[29] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 1106–1120, 2021.

[30] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 1093–1108, 2020.

[31] S. Van Schaik, K. Razavi, B. Gras, H. Bos, and C. 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.

[32] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jimenez, and M. Casas, "Exploiting page table locality for agile TLB prefetching," *Proceedings - International Symposium on Computer Architecture*, vol. 2021-June, pp. 85–98, 2021.

[33] D. A. Weikle, S. A. McKee, and W. 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*, pp. 2–12, 1998.

[34] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware TLBs," *Proceedings - International Symposium on Computer Architecture*, pp. 698–710, 2019.

[35] J. Zhang, W. Jia, S. Chai, P. Liu, J. Kim, and T. Xu, "Direct Memory Translation for Virtualized Clouds," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 2, pp. 287–304, 2024.