Predicting Execution Times With Partial Simulations in Virtual Memory Research: Why and How

Mohammad Agbarya* Idan Yaniv*
Technion – Israel Institute of Technology

Jayneel Gandhi
VMware Research

Dan Tsafrir
Technion & VMware Research

***Abstract*—Computer architects frequently use cycle-accurate simulations, which incur heavy overheads. Recently, virtual memory studies increasingly employ a lighter-weight methodology that utilizes partial simulations—of only the memory subsystem—whose output is fed into a mathematical linear model that predicts execution runtimes. The latter methodology is much faster, but its accuracy is only assumed, never rigorously validated.**

We question the assumption and put it to the test by developing Mosalloc, the Mosaic Memory Allocator. Mosalloc backs the virtual memory of applications with arbitrary combinations of 4KB, 2MB, and 1GB pages (each combination forms a "mosaic" of pages). Previous studies used a single page size per execution (either 4KB or 2MB) to generate exactly two execution samples, which defined the aforementioned linear model. In contrast, Mosalloc can generate numerous samples, allowing us to test instead of assume the model's accuracy. We find that prediction errors of existing models can be as high as 25%–192%. We propose a new model that bounds the maximal error below 3%, making it more reliable and useful for exploring new ideas.

> *"The phenomena surrounding computers are deep and obscure, requiring much experimentation to assess their nature." (A. Newell and H. A. Simon)*

I. Introduction

In recent years, more and more virtual memory studies abandon the traditional methodology of simulating the entire CPU and instead use *partial simulations* of only the virtual memory subsystem. The main reason for this trend is the increasing sizes of modern workloads in terms of instruction count and memory footprint. Evaluating the performance of such workloads with full ("cycle-accurate") simulations might take weeks, if not months, and thus might not be feasible [4], [7], [8], [21], [29], [30], [39], [59], [60], [66]. Partial simulations are 100x–1000x faster than full simulations, but they have an inherent drawback: they do not report application runtime, the metric that ultimately reflects the processor's performance. Instead, they output performance metrics specific to the virtual memory subsystem, notably, the number of TLB misses or the latency of walking the page table ("walk cycles"). Section II further motivates the use of partial simulations in virtual memory research.

To overcome this limitation, the aforementioned studies developed simplistic linear models that predict the runtime based on the partial simulations output, as outlined in Figure 1. Importantly, the models are tied to a given workload executing on a given processor. Namely, if workload W executes on processor P while experiencing a TLB miss rate of M (where M can be the output of a partial simulation), then the runtime of workload W on processor P is assumed to be a linear function: $R^P_W(M) = \alpha \cdot M + \beta$. Notably, P is some specific commercial CPU (e.g., Intel Xeon E5-2420), and the parameters α and β are fitted against one or two real (R,M) pairs that are measured using P's performance counters when W executes. Relying on a specific, real processor P in this way allows researchers to entirely refrain from running *any* full simulation.

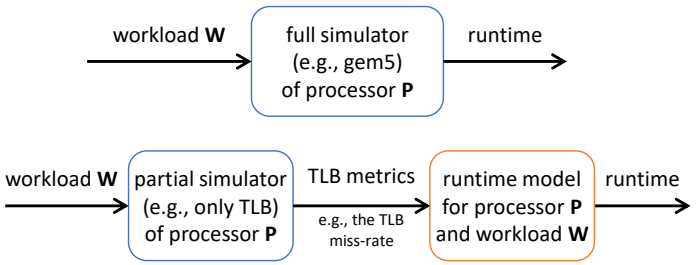

Figure 1: Partial vs. full (cycle-accurate) simulation.

Although virtual memory studies often rely on the above runtime models, they *assume*—rather than validate—the accuracy of their models. Some studies additionally neglect to clearly document their models, thereby hampering reproducibility. We spent many hours discussing the models with the authors of previous studies, uncovering the exact specifications and missing details. Our first contribution is therefore a comprehensive, detailed survey of all existing runtime models (Section III) and their limitations (Section IV).

We find that it was impossible for previous studies to validate their models, as their data was seemingly limited: at most two (R,M) execution points measured when the memory of W is backed by either 4KB or 2MB pages. Because these two points are used for fitting, no additional data remains for validation. Our second contribution is observing that such additional data can be obtained by mixing pages of different sizes. To this end, we design and implement Mosalloc, the "mosaic memory allocator", which mosaicks pages of different sizes into one contiguous virtual address space (Section V). Mosalloc is publicly available as an open-source library [2].

We apply Mosalloc on a set of memory-intensive workloads running on different Intel microarchitectures and obtain multiple new data points. Using this data, we show the previous models might deviate from true runtimes considerably, by up

* Both authors contributed equally to this work.

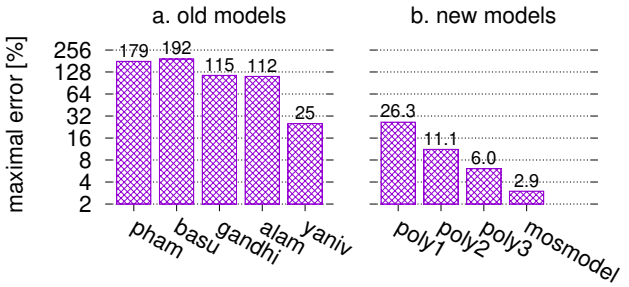

Figure 2: Preexisting models produce errors as high as 25%–192%. Mosmodel bounds the maximal error below 3%.

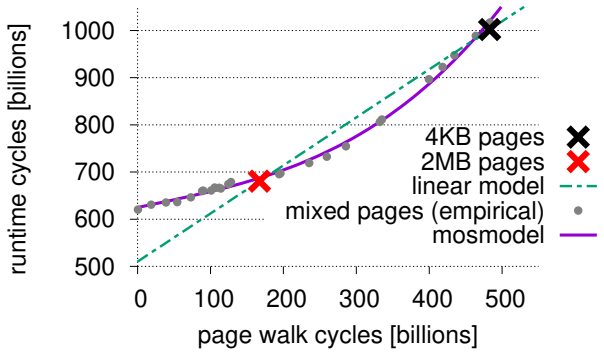

Figure 3: While the linear model is not able to predict the performance of spec06/mcf on our SandyBridge platform, Mosmodel is accurate enough (maximum error $< 2\%$).

to 25%–192% (as summarized in Figure 2a and discussed in detail in Section VI).

Our third contribution is developing a new runtime model, Mosmodel, which is 1–2 orders of magnitude more accurate than preexisting models (Section VII). Mosmodel improves upon its predecessors in three respects. First, it models many more execution points than just two with the help of Mosalloc. Second, it accommodates a new empirical observation—intuitive in retrospect—that arises from systematically using Mosalloc: CPUs may become increasingly effective in alleviating TLB misses when miss frequency drops and approaches zero (Figure 3). Polynomials of degree 1 (linear lines) and 2 (parabolic) are not flexible enough to model this observed behavior (Figure 2b), so we define Mosmodel to be a polynomial of degree 3.

The third difference between Mosmodel and its predecessors is accommodating another new empirical observation exposed by Mosalloc: runtimes of different workloads are predicted better by different performance metrics, either TLB misses, or TLB hits, or the aggregated amount of cycles spent on walking the page tables. Mosmodel consequently utilizes the three corresponding variables, selecting the most suitable on a per workload basis. As Mosmodel succeeds to bound the maximal relative error below 3%, it allows researchers to more reliably enjoy the benefits of partial simulations.

Architects are usually primarily interested in their partial simulators (which, e.g., test some new design), whereas we exclusively focus on the complementary runtime models and completely ignore the partial simulators. Here is why. Recall that, by definition, the models R_W^P that we study are tied to some specific, real processor P. The **goal of this study** is to address a single key question: how accurate is R_W^P in predicting the runtime of its own P? This question is independent of any partial simulator—it can only be answered by executing workloads (W) on P and measuring their runtime. Crucially, it makes sense for researchers to use the model R_W^P for exploring new architectural designs (as in Figure 1) only if the answer to our key question ("how accurate is R_W^P in predicting P") is "reasonably accurate." If R_W^P fails to predict even its own original P, then it is clearly wrong to assume that R_W^P can predict a modified P whose (modified) virtual memory subsystem is being simulated. Thankfully, Mosmodel is reasonably accurate.

II. Motivation

A. Why Trust Partial Simulation Predictions?

The premise underlying the partial simulation methodology is the following. Based on empirical measurements of the system, it is possible to devise a mathematical model capable of predicting the behavior of the *full* system, from the behavior of only a *subset* of the system's components. The validity of all the virtual memory papers that employed partial simulations [4], [7], [8], [21], [29], [30], [39], [59], [60], [66] explicitly resets on this premise—if it is invalid, then their outcome is invalid as well. A question that sometimes arises when discussing the issue of utilizing partial simulations in the context of virtual memory research is: Why should we believe the predictions of some runtime model (which we do not typically "understand"), whose input is representative of only a part of the system, given that surely there are nontrivial interactions between this part and the other parts? Because this question arises, we believe that it is worth considering the partial simulation methodology in general—beyond the context of computer architecture—and point out that it is in fact a standard scientific methodology, routinely used by scientists to predict the behavior of complex systems fast.

A recent high-profile example of scientists relying on this premise is the particle physicists at the ATLAS detector in CERN, who proved the existence of the Higgs boson. These physicists regularly employ the full and partial simulation methodology to explore physical theories [1]. The full ATLAS simulator is slow, limiting its applicability. Partially simulating it yields faster results but only outputs a limited set of metrics. The ATLAS physicists therefore introduced mathematical models that extrapolate the full simulation output. In particular, they developed ATLFAST-I, a parameterized detector model that makes use of (is fitted against) measured data. Fitting against empirical measurements does not reflect any qualitative understanding. Yet empirically, the ATLFAST-I model works: it serves as a fast replacement for the full ATLAS simulator, predicting the full detector readout with 10%–20% accuracy.

As highlighted in Table 1, the partial simulation methodology employed by computer architects is quite similar to the one

methodology	*aspect*	*virtual memory research*	*ATLAS experiment*
full simulation	models	all CPU components, cycle by cycle	collisions, their products, and their interaction with the detector
	outputs	application runtime	detector readout
partial simulation	models	parts of virtual memory subsystem, e.g., TLB	particle collisions only
	outputs	virtual memory performance metrics, e.g., TLB misses	type, momenta, and energy of particles generated in collisions
	requires	runtime model (thus far: linear)	parameterized detector model
	validated	**no**	yes

Table 1: Using partial simulations to model a full system is standard in scientific experiments. The partial simulation methodologies used in virtual memory research and in the ATLAS experiment are analogous. But only the latter has been validated.

employed by partial physicists, with one important difference: thus far, our community did not assess the accuracy of the runtime models it employs.

B. Pros and Cons of Partial Simulations of Virtual Memory

Computer architects regularly use full (cycle-accurate) simulators to estimate the performance of new CPU designs without having to build costly hardware prototypes [20], [25]. Full simulations run at rates of several kilo instructions per second, possibly millions of times slower than executing the application on a physical CPU. As contemporary workloads may run for several minutes to hours, fully simulating them will take weeks, if not months and thus might not be feasible. These high overheads severely limit the size of workloads and number of hardware configurations that, realistically, can be simulated to explore the design space. Others have also observed that overreliance on full (cycle-accurate) simulators may result in overfitting to a particular design point [55]. Another weakness of full simulation is the high development effort it requires from researchers. For example, the popular gem5 simulator contains 1.5 million lines of code [16], [17], so understanding and modifying the code of this complex full simulator might be challenging.

Full simulations are slow to execute because they capture the entire system with great detail. Recent virtual memory studies therefore opted for partial simulations, which reproduce only the virtual memory subsystem, e.g., the TLB and hardware page walker [4], [7], [8], [21], [29], [30], [39], [59], [60], [66]. For example, several recent studies developed partial simulators based on BadgerTrap, a Linux kernel instrumentation tool for tracing TLB misses [4], [8], [28], [29], [30], [39]. BadgerTrap slows down workloads by 2x–40x, so it allows much faster simulation than gem5.

Partial simulators typically output the TLB hit and miss rates but they can also report the TLB miss latency if they simulate the memory hierarchy. In the x86-64 architecture, TLB misses are served by four consecutive reads from the hierarchical page table [5], [36]. These four memory references are non-overlapping because each page table entry is accessed after reading the entry in the previous level. Calculating the page walk latency thus requires simulating the four references to the memory hierarchy (L1, L2, L3 caches, and DRAM) and summing their latencies. Page walk caches accelerate the page walk by caching parts of the page table, so partial simulators should also incorporate them to accurately calculate the number of walk cycles [7], [12], [13], [66]. Simulating the memory hierarchy and page walk caches (PWCs) is indeed more complicated than simulating the TLB alone, but is still faster and simpler than simulating the entire CPU.

Partial simulators are faster and easier to develop, but they have an inherent drawback—they cannot report the application runtime, which is the metric computer architects typically use to determine the processor performance. To overcome this limitation, most partial simulation studies introduced linear models that predict the runtime based on the number of TLB misses, as illustrated in Figure 1. All existing models are surveyed in Section III.

Considering the wide adoption of linear models for predicting runtime, we initially ask: are the linear models accurate enough? Previous studies neither addressed nor acknowledged this question. In this study, our first contribution is developing a new methodology that allows us to answer this question. We find that the somewhat disappointing answer is that the models might deviate considerably by up to 25%–192% from real runtimes, as shown in Figure 2a. Arguably, such errors are unacceptable nowadays, when the average performance improvement rate of processors is around 10% per year (page 3 in [32]). Runtime prediction should be, say, an order of magnitude more accurate, within 1%, to allow for a more reliable computer architecture research.

C. No Simulation Methodology is a Silver Bullet

The aforementioned high errors of existing runtime models seemingly indicate that the partial simulation methodology is unreliable, which implies that the full simulation methodology is preferable for computer architecture research, despite its high cost. We contend that this is not the case for two reasons. First, because in this paper we develop a runtime model that is significantly more accurate (Figure 2b). The second reason is that the full simulation methodology, as commonly practiced, suffers from notable drawbacks too, as outlined next. As these drawbacks might cast a shadow on the reliability of full simulations, it is not that one methodology is strictly preferable to the other.

One drawback, as previously noted, is that full simulations of real-world workloads are often too lengthy to be feasible. Researchers therefore usually resort to sampling the instruction stream input to reduce simulation time. In particular, the common practice in virtual memory studies from the last decade is to use “blind sampling”, namely, to fast-forward a few billions of instructions of the workload and then to simulate another few billions [3], [14], [15], [19], [24], [48], [49], [56], [57], [58], [63]. (For comparison, typical workloads execute hundreds to thousands of billions of instructions.)

A main weakness of blind sampling is that it might be nonrepresentative, because it ignores the time varying behavior of real workloads. Indeed, the seminal SimPoint work measured an average simulation error of 80% for when blind sampling was applied [61].

Another drawback pointed out by the SimPoint study is that sampling is inherently ineffective for capturing relatively rare events, like L2 cache misses, even if using SimPoint's phase-aware sampling. Therefore, arguably, it is possible that applying sampling in virtual memory studies, whose focus is workloads that experience relatively infrequent events like TLB misses, might lead to unreliable full simulation results. Future research may perhaps develop reliable sampling methods in the context of virtual memory research. But as things currently stand, *both* the full and the partial simulation methodologies require validation in this context, and it is possible that both yield results that are not representative of real (rather than simulated, sampled) systems.

While sampling is typically necessitated for conducting full simulations in a reasonable time, the sampling component is in fact orthogonal to the type of simulation. Partial simulations can work on sampled application traces just like full simulations do. And indeed, several virtual memory studies that employed partial simulations also sampled their input to speed up the simulation [4], [21], [59], [66]. Validating the sampling technique for full simulations is thus an opportunity for accelerating partial simulations as well. Importantly, by definition, partial simulations can be much shorter than full simulations, and so they will continue to be a valuable tool for researchers to explore wider spaces of parameters and configurations, assuming they are validated.

D. Validating Runtime Models

Enhancing partial simulations with runtime models is a compelling idea, because it combines the speed of partial simulations with the ability to estimate the bottom-line performance. Alas, to our knowledge, no study has proven that the previously proposed linear models are indeed capable of accurately predicting the runtime. Validating runtime models, just like validating any scientific model, requires experimental data to compare against the model predictions. The problem is that current research has very little data: two points, measured when the application uses either 4KB or 2MB pages. Since these two points are used to fit the linear models, we cannot use them to validate the models.

Proper validation of the linear models requires more data than the two points collected for 4KB and 2MB pages. The problem is that x86-64 processors support only three page sizes, which might suggest that it is possible to obtain only one more experimental data point. We speculate that this could be the reason previous studies never attempted to validate the models they used. Our main insight is that multiple empirical points are possible to obtain by mixing pages of various sizes, in a controlled manner, when backing the memory address space of the application at hand. There are no allocators that support such a functionality. We thus designed and implemented Mosalloc, a new memory allocator, which allows users: (i) to back the address space of applications with an arbitrary mix of pages of different sizes, and (ii) to control and systematically vary the number and placement of these pages. Mosalloc stands for "Mosaic Memory Allocator". We use the term "mosaic" because the allocator is mosaicking pages of different sizes into one contiguous virtual address space. Section V describes the design and implementation of Mosalloc.

notation	*description*
R	runtime: num. of unhalted application execution cycles
H	num. of translations that missed on L1 TLB but hit on L2 TLB
M	num. of translations that missed on both L1 TLB and L2 TLB
C	walk cycles, spent on walking the page table upon TLB misses

Table 2: Performance metrics utilized by previous studies to define their (linear) models, which they employed to complement their partial simulations. The metrics were collected on real CPUs on a per-benchmark basis.

We used Mosalloc to run several benchmark workloads under dozens of mixed memory layouts that were not possible thus far. We collected the performance statistics of these runs on three different x86-64 platforms to obtain dozens of experimental samples for each workload on each processor. Unlike previous studies, our datasets contain many more than just two experimental samples. This new data obtained via Mosalloc enables us to validate the linear runtime models [4], [7], [8], [21], [29], [30], [39], [59], [60], [66], as reported in Section VI.

We emphasize that the scope of this study is validating runtime models and not simulators. Simulators and runtime models are of course connected, as the input for the runtime model is obtained through partial simulation (see Figure 1), and so accurate simulations are a prerequisite for accurate predictions. Still, runtime models stand in their own right regardless of simulations, and the model accuracy is independent of the simulator accuracy.

III. Existing Runtime Models

The runtime models proposed in prior virtual memory research are all linear. Namely, the models assume that runtime is a linear function of the virtual memory metrics as listed in Table 2: the TLB hit rate, the TLB miss rate, or the number of walk cycles (that is, the number of CPU cycles spent during page table walks). We name these models after their authors: the Basu, Pham, Gandhi, Yaniv, and Alam models. We spent many hours discussing the models with their authors and making sure we represented their work accurately. Each model assumes a somewhat different linear form but all are defined by one or two data points collected via the performance monitoring unit (PMU) of the processor. The PMU consists of a set of hardware performance counters: special-purpose registers that are able to store the counts of processor events, e.g., the number of TLB misses or the number of branch mispredictions. All previous work collected data on Intel platforms, as only Intel PMUs provide the ability to measure the walk cycles (e.g., events 0x0408, 0x0449 in the SandyBridge microarchitecture [37]).

The Basu Model [8], [60] was the first linear model for estimating runtimes. The model assumes that an application runtime R is a linear function of M, the number of TLB misses while the application is running (M is the output of a partial simulation):

$$R = \alpha \cdot M + \beta \quad .$$

The model parameters α, β are fitted to empirical data measured when the application runs on a real machine. That is, if M_{4K}, C_{4K}, and R_{4K} are the measured number of TLB misses, cycles spent on page table walks, and the total execution cycles when using 4KB pages, respectively, then:

$$\alpha = \frac{C_{4K}}{M_{4K}} \quad , \quad \beta = R_{4K} - C_{4K} \quad .$$

Basu model is thus the linear curve that passes through the following two points in the two-dimensional (M,R) space:

$$(0, R_{4K} - C_{4K}) \quad , \quad (M_{4K}, R_{4K}).$$

The Basu model is based on two simplistic assumptions. The first assumption is that the ideal runtime, i.e., the execution time when the L2 TLB never misses, is $\beta = R_{ideal} = R_{4K} - C_{4K}$. The mathematical interpretation is that servicing TLB misses stall the application from progressing, so eliminating all TLB misses shortens the runtime R_{4K} by the cycles spent on page walks, as counted by C_{4K}. The second assumption is that the latency of servicing a TLB miss (in units of cycles) is constant and equal to the average latency $\alpha = C_{average} = \frac{C_{4K}}{M_{4K}}$. Unfortunately, these two assumptions do not hold in practice, and the maximum relative error of the Basu model is 192%, as reported in Section VI.

While Basu et al. considered the runtime overestimation favorably, we argue that it is a serious weakness of their model. Overestimating the runtime can be considered as conservative approach when assessing the benefits of a new architectural design. For example, Basu et al. used their model to bound the runtime gains from their proposed direct segments design, promising that the true runtimes are lower than those predicted by their model. But a model that sometimes overestimate runtimes is problematic when comparing several new architectural designs. Computer architects, who are required to choose between direct segments and another competing design, should know the accurate runtime benefits–rather than lower bounds–of these designs.

The Gandhi Model [29], [30], [39] also assumes:

$$R = \alpha \cdot M + \beta \ ,$$

like the Basu model, but defines the parameters differently:

$$\alpha = \frac{C_{4K}}{M_{4K}} \quad , \quad \beta = R_{2M} - C_{2M} \ .$$

Gandhi et al. recognized that subtracting the page walk cycles from the total execution cycles may be inaccurate when the page table walks overlap with other processor stalls. They believed that calculating the ideal runtime β from the 2MB pages configuration will minimize this inaccuracy and fix the shortcomings of the Basu model. Unfortunately, the results in Section VI demonstrate that the Gandhi model is still not accurate enough, with prediction errors as high as 115%.

The Pham Model [21], [59] assumes that every cycle spent on address translation, either when the CPU misses in the L1 TLB and searches the L2 TLB or when the CPU misses in the L2 TLB and walks the page table, is directly added to the overall runtime:

$$R = 7 \cdot H + C + \beta \ .$$

The number of L2 TLB hits H is multiplied by 7 because this is the L2 TLB access latency reported by Intel [38]. Similarly to the Basu model, the parameter β is the "execution time if virtual memory was completely free," i.e., if $C = 0$ and $H = 0$. β is calculated by:

$$\beta = R_{4K} - C_{4K} - 7 \cdot H_{4K} \ .$$

The primary flaw of the Pham model lies in its naive assumption that the CPU stalls when it translates virtual addresses. In practice, however, modern CPUs are able to execute multiple independent instructions simultaneously (superscalar and out-of-order) without waiting for the page walk to be completed, thereby hiding a large amount of the address translation latency. This problem was also recognized by other researchers as well [49] This unrealistic assumption underlying the Pham model may lead to optimistic (that is, lower) predicted runtimes for some workloads. Our full numbers (not given in this paper due to space constraints) confirm that the Pham model predicts optimistic runtimes for *all* tested workloads and machines. Section VI shows that the relative error of the Pham model may be as high as 179%.

The Alam Model [4] takes the number of table walk cycles C as an input:

$$R = C + \beta \ ,$$

and defines the single parameter β similar to Gandhi:

$$\beta = R_{2M} - C_{2M} \ .$$

Alam et al. used this model to estimate the performance of their newly-proposed DVMT (Do-It-Yourself Virtual Memory Translation) design. The runtime of the DVMT configuration is calculated according to:

$$R_{DVMT} = C_{DVMT} + \beta \ .$$

Since the number of walk cycles C_{DVMT} cannot be measured directly but only simulated, Alam et al. proposed to estimate it through:

$$C_{DVMT} = C^{sim}_{DVMT} \cdot \frac{C_{4K}}{C^{sim}_{4K}}.$$

In other words, Alam et al. compensated for the simulator inaccuracy by scaling the simulation output by some factor. We evaluated the Alam model on real data measured on our three experimental platforms, so we did not need to apply this scale factor. We found that the maximum relative error of the Alam model is 111%.

The Yaniv Model [66] does not assume that walk cycles are simply added to the runtime, but instead assumes that the overhead is the walk cycles multiplied by some factor α:

$$R = \alpha \cdot C + \beta \ ,$$

The parameter α can be thought of as the page-walk penalty factor. For example, $\alpha = 0.7$ means that every cycle spent on page table walks slows the application by 0.7 cycles. The model is thus more flexible than the Pham model as it introduces a second parameter for the slope, α, in addition to β, which, as before, is the ideal runtime when virtual memory incurs no overhead. This flexibility relaxes the unrealistic assumption underlying the Pham model that page table walks completely halt the progress of the application. The model parameters α, β correspond to the linear curve that passes through two points, (C_{2M}, R_{2M}), (C_{4K}, R_{4K}), which are measured when the application uses 4KB and 2MB pages, respectively. Section VI shows that the Yaniv model predictions deviate from true runtimes by up to 25%.

IV. Modeling Assumptions and Limitations

Recall that the preexisting models surveyed in the previous section are tied to a given workload W executing on a given processor P. (We denoted R rather than R_W^P only for brevity.) Executing the same workload W on different processors P_1, P_2 typically produces different performance counter values of R, H, M, C. The resulting model $R_W^{P_1}$ would thus be different than $R_W^{P_2}$ because the model parameters are fitted to the performance counter data. Similarly, two workloads W_1, W_2 running on the same processor P typically yield different performance counter values and hence different models. The conclusion is that modifying the workload source code or even linking the compiled object files in a different order [52] require running the new workload to collect the performance counters (whose values typically change) and then recomputing the model parameters based on the new values.

Considering that all runtime models (past and new) are processor-specific, our study asks: can these models actually serve the purpose for which they were invented, that is, predicting the performance of *newly*-proposed processor designs? Namely, is it reasonable to estimate the performance of a new processor $\bar{P}$ with a model R_W^P built for another processor P? Importantly, all the relevant previous studies assumed that the answer is positive [4], [7], [8], [21], [29], [30], [39], [59], [60], [66]. We contend that a necessary condition for this assumption to hold is that R_W^P accurately predicts the runtime of P, the original processor for which the model was built. Conversely, if R_W^P is not accurate enough for describing P, then we should not expect the model to predict the runtime of $\bar{P}$ accurately. A main goal of this study is to assert this necessary condition by measuring the accuracy of preexisting models with respect to their associated processors.

Another assumption underlying the partial simulation methodology is that P and $\bar{P}$ differ in their virtual memory subsystem only while the other processor subsystems of P and $\bar{P}$ are identical. By definition, a runtime model whose inputs are virtual memory related metrics (H, M, C) accounts for part of the overall performance. The model abstracts from the potentially nontrivial interactions between the virtual memory subsystem and other system components. For example, increasing the size of the processor data caches may not affect TLB miss rate but will frequently affect the runtime. Similarly, more sophisticated out-of-order execution could be more effective in hiding TLB miss latency. It is probably safe to assume that it is impossible to capture such architectural differences via a model that merely factors virtual memory related performance metrics. (Conceivably, it may perhaps be possible to develop a mathematical formula that predicts runtime in a manner that is not application- or architecture-specific, by having it utilize many more variables that correspond to many more architectural events; this is outside our scope of work.)

As noted, the partial simulation methodology is common in other scientific fields; Section II-A gives the ATLAS experiment as an equivalent example.

V. Mosalloc: Design and Implementation

We created Mosalloc, a memory allocator that serves memory requests with a user-defined mixture of standard pages and hugepages. Mosalloc is implemented as a dynamic library that is loaded before glibc and hooks all memory requests made by an application. First, Mosalloc intercepts malloc [44] requests by hooking the morecore function, which malloc calls when it needs to extend the heap. Second, Mosalloc intercepts direct invocations of brk [42], mmap [46] and munmap [47], the primary memory system calls in Linux, by overriding their glibc wrapper functions. To apply Mosalloc, the user should link the library to the application at run-time using the LD_PRELOAD [43] environment variable. Note that Mosalloc is an independent library so it does not require modifying the existing source code or rebuilding the application. Additionally, Mosalloc is implemented in user-space and does not require kernel modification.

Mosalloc allows the user to back the address space of applications with arbitrary combinations of page sizes. To accomplish this, Mosalloc manages three pools that serve the three types of memory requests in Linux: (1) brk calls, (2) anonymous mmap calls, and (3) file-backed mmap calls. The user should specify the layout of the brk pool and the layout of the anonymous mmap pool through a set of environment variables, which Mosalloc takes as an input. The brk and anonymous mmap pools can mix pages of different sizes, as outlined in Figure 4. Mosalloc allocates these pools via the mmap system call with the relevant flags (MAP_HUGETLB, MAP_HUGE_2MB, and MAP_HUGE_1GB). The file-backed pool is backed only with 4KB pages because Linux does not support file-backed mmap calls with hugepages; these mmap calls are served from the page cache, which Linux manages only with 4KB pages [46], [54].

The Heap Pool serves morecore requests and direct calls to the brk and sbrk system calls. This pool effectively replaces the original heap allocated by the operating system. When glibc malloc needs to extend the heap, it calls morecore, which in

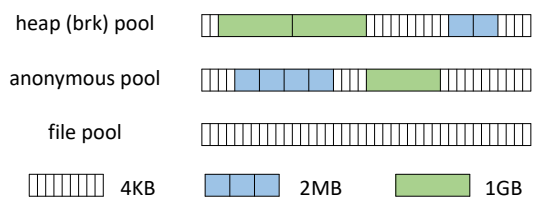

Figure 4: Mosalloc forwards user memory requests to 3 separate memory pools. The heap and anonymous mapping pools are backed by user-specified combinations of 4KB, 2MB, and 1GB pages. The file mapping pool is backed by 4KB pages only.

turn calls sbrk. The sbrk and brk system calls allocate and deallocate memory by changing the location of the heap top (the "program break"). When glibc is loaded, it first calls sbrk(0) to get the current address of the heap top. Mosalloc intercepts this call and returns the current address of its heap pool. Further brk and sbrk calls by the program will refer to this returned address, so they will automatically allocate and deallocate memory on the heap pool.

The Anonymous Mapping Pool serves mmap calls that specify the MAP_ANONYMOUS flag [46]. Memory allocations in this pool are served according to the "first fit" algorithm. We chose this algorithm because it performs better than the alternatives of "best fit" and "worst fit" in terms of runtime complexity and memory utilization [62]. Similarly to the heap pool, Mosalloc frees memory only from the top of the anonymous mapping pool. While this design simplifies the pool management, it may lead to memory fragmentation compared to glibc munmap, which immediately reclaims memory regions. We measured the additional memory consumption to be less than 1% for our tested workloads. We leave the development of better, more efficient memory management algorithms for future work.

A. Work Related to Hugepages

Explicit Hugepage Support was the first mechanism Linux offered for user code that wishes to back its memory with hugepages. This mechanism relied on the hugetlbfs virtual file system—a pool of hugepages [22], which the root user can reserve at boot-time or at run-time (if there is enough contiguous memory). User code can explicitly request to use these reserved hugepages by invoking the mmap system call with the MAP_HUGETLB flag along with either the MAP_HUGE_2MB or MAP_HUGE_1GB flags [46]. Mosalloc differs in that it serves memory allocation requests with hugepages transparently, without modifying the user code.

Transparent Hugepages (THP) was introduced in Linux 2.6.38 [23], [53]. As its name suggests, this feature provides transparent allocation of hugepages to user applications. "Transparent" means that the user does not have to modify and rebuild the application code. In fact, THP does not even require to link the binary with some library at runtime, as required with Mosalloc. THP is limited compared to Mosalloc because it: (1) does not allow the user to specify the exact placement of hugepages in the application address space, (2) supports only 2MB pages and not 1GB pages, and (3) interferes with the application execution by dynamically promoting/demoting hugepages, which might cause significant overheads [41].

Libhugetlbfs is a user-space library that allows applications to back their memory with a single size of hugepages [50]. Similar to Mosalloc, applications that wish to use libhugetlbfs are not required to modify their code, but instead load this library at runtime by setting the LD_PRELOAD environment variable [43]. Additionally, the user should specify the requested hugepage size, either 2MB or 1GB, in the HUGETLB_MORECORE environment variable. Libhugetlbfs works by hooking the morecore function, which malloc calls when it needs to extend the heap. Libhugetlbfs implements morecore by allocating a hugepage-backed block at the heap top.

Libhugetlbfs has several limitations that motivated us to develop Mosalloc. First, libhugetlbfs backs the address space of applications uniformly, using either 2MB or 1GB pages, and does not allow mixing pages of different sizes in a controlled manner. Second, libhugetlbfs intercepts only malloc calls, so it does not support workloads that allocate their memory with the mmap or brk system calls, e.g., the graph500 benchmark. Third, workloads that use allocators different from glibc, e.g., Hoard [10], [11] or TCMalloc [31], cannot use libhugetlbfs to alter their memory allocations because these allocators do not provide the morecore hook. Finally, libhugetlbfs fails to intercept all allocation requests because malloc sometimes gets more memory without using morecore, as we explain later in Section V-C.

B. Mosalloc Contributions

Mosalloc is the first tool that allows user applications to back their memory with a predefined combination of hugepages. As a user-space library, Mosalloc has the advantage that it is portable across Linux kernel versions. Similar to libhugetlbfs [50], Mosalloc can be loaded dynamically with existing binaries to transparently back their memory with hugepages. Unlike libhugetlbfs, Mosalloc is not based only on glibc morecore hook and is able to support applications and libraries that manage their memory with mmap and brk. In other words, Mosalloc is more portable than libhugetlbfs because it intercepts all POSIX system calls that allocate memory (rather than just malloc), so it can work in principle on any POSIX-compliant operating system and/or libc implementation, e.g., musl libc [26] (although we tested it only on Linux with glibc). Additionally, Mosalloc fixes a bug in libhugetlbfs, which does not intercept malloc requests that call mmap directly, as explained in the next section. Finally, Mosalloc is somewhat simpler to use than libhugetlbfs because it does not require mounting the hugetlbfs file system.

Mosalloc may have broader use cases beyond performance prediction for computer architects. For example, high-end users may optimize the performance of their Linux applications by using Mosalloc to back memory regions that suffer from TLB

misses with hugepages. We released Mosalloc publicly, hoping that other researchers and engineers will find it useful [2].

C. Implementation Challenges

The main technical challenge in implementing Mosalloc was guaranteeing that it hooks *all* memory allocation requests made by the application. Naively preloading a library that overrides the memory allocation functions malloc, brk, mmap, and munmap does not work, because these functions are implemented in the same library — glibc. Consequently, mmap calls from malloc are statically linked to the mmap address at compile-time. Mosalloc needs to eliminate such mmap calls completely because it cannot hook them at run-time. A possible solution would be modifying glibc malloc to call Mosalloc instead of mmap. Unfortunately, modifying glibc is non-trivial because glibc auto-generates large parts of its source code. We therefore used two "tricks" to disable mmap calls from malloc.

First, malloc calls mmap directly when the requested block is larger than MMAP_THRESHOLD (defaults to 128KB) and bypasses the morecore function, which serves memory requests smaller than MMAP_THRESHOLD. Since these direct calls to mmap cannot be hooked, Mosalloc disables them by setting the M_MMAP_MAX parameter to 0 through mallopt [45]. Libhugetlbfs uses the same technique to force malloc not to allocate memory with mmap.

Second, malloc calls mmap directly when it detects a lock contention by concurrent memory allocations from multiple threads. In that case, malloc allocates new memory regions, called "arenas", that will serve the contending memory requests concurrently and reduce the allocation latency. The new arenas are allocated with mmap rather than morecore, which means they cannot be hooked. Mosalloc limits glibc to use only one arena through the M_ARENA_MAX parameter of mallopt. Libhugetlbfs does not use the same technique so it does not allocate all application memory with hugepages. We consider this behavior a bug of libhugetlbfs.

VI. Existing Models Are Inaccurate

A. Benchmarks and Platforms

We tested Mosalloc on three Intel platforms, described in Table 3, to examine processors with different virtual memory designs. Table 4 displays the TLB parameters of five recent Intel processor generations. The TLB of SandyBridge and IvyBridge contained 512 entries, which then doubled in Haswell and tripled in Broadwell [38]. The number of TLB entries for 2MB pages has also increased substantially, from 32 in SandyBridge to 1536 in Skylake. Furthermore, starting at Broadwell, Intel processors are equipped with a second page table walker to handle TLB misses parallel with the primary one. We disabled hyper-threading to tune our machines for maximum performance; Intel splits the L1 and L2 TLB entries between logical cores when hyper-threading is enabled. Similarly to previous studies, we did not test AMD or ARM processors because they cannot measure "walk cycles" with their PMU.

We tested Mosalloc on a set of workloads from several benchmark suites, which are listed in Table 5. We examined

generation	*processor (cores x sockets)*	*main memory*	*L3*
SandyBridge	1.9GHz Xeon E5-2420 (6Cx2)	96GB/1.6GHz	15MB
Haswell	2.1GHz Xeon E7-4830 v3 (12Cx2)	128GB/1.6GHz	30MB
Broadwell	2.2GHz Xeon E7-8890 v4 (24Cx4)	512GB/2.4GHz	60MB

Table 3: All the machines we use run Ubuntu 18.04.3 LTS (Linux 4.15) and are tuned for maximum performance (TurboBoost on and hyper-threading off in BIOS). All CPUs have 32 KB L1d, 32 KB L1i, and 256 KB L2 caches per core.

		L1 TLB entries			*L2 TLB entries*			*page*
generation	*year*	*4KB*	*2MB*	*1GB*	*4KB*	*2MB*	*1GB*	*walkers*
SandyBridge	2011	64	32	4	512	0	0	1
IvyBridge	2012	64	32	4	512	0	0	1
Haswell	2013	64	32	4	1024	shared	0	1
Broadwell	2014	64	32	4	1536	shared	16	2
Skylake	2015	64	32	4	1536	shared	16	2

Table 4: TLBs have grown in recent Intel microarchitectures.

only TLB-sensitive workloads, whose performance varies by at least 5% when backed with 1GB pages. (Mosalloc can be applied to any Linux x86-64 executable, but there is no point in testing workloads whose performance is independent of the memory layout.) We also limit our investigation to short benchmarks, running for less than 10 minutes on our Broadwell machine, to allow us to measure each benchmark under dozens of virtual memory layouts. Each workload is run for several repetitions until the variation in runtime (the ratio of the standard deviation to the mean runtime) is less than 5%. The error bars are not shown in all figures to avoid cluttering. Each run is bound to the cores and memory of a single socket to minimize NUMA effects. Multi-threaded workloads were given all cores of the processor to which they are bound.

B. Selecting Memory Layouts

Mosalloc allows users to back address spaces with arbitrary combinations of regular pages and hugepages. But it does not *find* combinations that yield helpful data. As our goal is to validate runtime models over a wide range of inputs, we require memory layouts that produce distinct data points spread across the (H,M,C) space (regardless of if they are "realistic" layouts, which is irrelevant). We thus develop an algorithm that finds such points using three layout-exploration heuristics: growing window, random window, and sliding window. A window is a contiguous memory region covered with 2MB hugepages. A heuristic generates N+1 layouts for a given N.

Growing Window This heuristic backs a growing part of the address space with hugepages. Let S be the address space size. For i=0,1,...,N, the i-th layout window starts at 0 and covers

suite	*cites*	*suite description & benchmark selection*
SPEC CPU2006	[33], [34]	single-threaded compute: mcf, omnetpp
SPEC CPU2017	[18], [35]	likewise: xalancbmk_s, omnetpp_s
Graph500	[6], [51]	compress+BFS graphs of size: 2/4/8 GB
GUPS	[40]	random read from array of size: 8/16/32 GB
XSBench	[64]	multithreaded Monte Carlo simul.: 4/8/16 GB
GAPBS	[9]	multithreaded kernels: BC, PR, BFS, SSSP on real-world graphs: twitter, road, web

Table 5: In each suite, we use all TLB-sensitive benchmarks.

$i \cdot S/N$ of the space. Thus, the first layout uses 4KB pages only, and the last layout backs the entire space with 2MB pages.

Random Window This heuristic generates each layout with a window that has a random length and start address.

Sliding Window This heuristic is more sophisticated. It (1) collects the workload's TLB miss trace with PEBS [37]; (2) identifies the smallest "hot region", namely, a contiguous segment that accounts for X percent of all TLB misses (when using 4KB pages) for a given X; (3) defines the hot region as the window of the first layout; and (4) slides this window in steps of $1/N$ of the hot region size, thus gradually backing a smaller part of the region with hugepages. Sliding is towards the low or high addresses depending on if the hot region is at the top or bottom of the address space, respectively.

We construct 54 layouts for each workload: 9 layouts with Growing Window (N=8), 9 with Random Window, and 9×4=36 with Sliding Window using four values of X (20%, 40%, 60%, and 80%). Sliding Window yields the most diverse outcome, because (1) it utilizes additional information to detect and focus on the hot region (the TLB miss trace), and because, (2) empirically, for most workloads, TLB misses are mostly concentrated in a relatively small memory region. For example, 80% of the TLB misses of graph500/2GB originate from its heap's highest 80MB. Thus, in this case, a random layout would typically either entirely back or entirely miss this region, performing similarly to either "all 2MB" or "all 4KB" layouts, respectively. For this reason, random sampling is less effective.

C. Fitting Models and Measuring Prediction Errors

We run each workload W on each processor P with each of the i=1,2,...,54 memory layouts. We measure the runtime R_i and metrics (H_i,M_i,C_i) and thus acquire many more samples than just the two used by previous models. We can now assess the accuracy of the *existing* models (surveyed in Section III) by calculating the maximal absolute relative error:

$$maxErr(W,P) = \max_{1 \le i \le 54} \left| \frac{R_i - \hat{R}_i(H_i,M_i,C_i)}{R_i} \right|, \quad (1)$$

and the associated geometric mean:

$$geoMeanErr(W,P) = \prod_{1 \le i \le 54} \left(\left| \frac{R_i - \hat{R}_i(H_i,M_i,C_i)}{R_i} \right| \right)^{\frac{1}{54}}, \quad (2)$$

where R_i and $\hat{R}_i$ are the measured and predicted runtimes, respectively. The minimal absolute error is zero for all models, as all pass through at least one experimental data point.

To build and test our newly proposed models (defined in Section VII), we employ two methods. The first fits and tests the models against all available data. Namely, for each W and P pair, we use all 54 samples to build the corresponding model and to measure its accuracy with Equations 1–2. We select the number of samples (54) to be big enough to adhere to the one-in-ten rule, which statisticians employ to keep the risk of overfitting low when conducting regression analysis [65]. Whereas our most complex model has 19 coefficients (third-degree polynomial with three variables; see Section VII-C), it utilizes Lasso regression that leaves only 5 nonzero coefficients or less, which is indeed ≥10x smaller than 54.

model	poly1	poly2	poly3	Mosmodel
maximal error	36.4%	19.1%	20.0%	4.3%

Table 6: Maximal cross validation errors (compare to Figure 2b).

The second method we use to build and test our models is K-fold cross validation [27]. This method combats overfitting by splitting the data into K disjoint equal-sized subsets, called "folds", such that K-1 folds serve as a training set (to fit the model parameters) and the remaining fold serves as the test set (to compare against the model's predictions and compute the errors). This procedure is conducted K times, such that each individual fold serves as the test set. We then compute the maximal error across all K test folds.

All results shown in this paper relate to the first method, which fits/tests against all the data. The exception is Table 6, which summarizes the maximal cross validation errors of our new models across all machines and workloads.

When comparing this table to Figure 2b, we see that cross validation errors are worse, but that Mosmodel still clearly outperforms the rest and yields a relatively low maximal error. (Cross validation is irrelevant to the preexisting models shown in Figure 2a, as they are entirely determined by one or two specific points and therefore cannot be trained.)

We favor fitting/testing against all data points over cross validation (while adhering to the one-in-ten rule to reduce the risk of overfitting), because our experience suggests that the former is more compatible with the highly-sensitive maximal error metric, in that it converges faster. In particular, when using cross validation, 54 samples were sometimes not enough, requiring us to use up to 100 points to achieve a low (≤5%) maximal error for Mosmodel.

D. Results and Discussion

Figures 5 and 6 present the maximal and geometric mean of the prediction errors, respectively, for all tested workloads and platforms. Our analysis henceforth focuses on maximal (rather than mean) errors, as we want to highlight the worst case and also refrain from bias caused by taking into account the 4KB and 2MB points, for which the linear model errors are zero. We find that all previously proposed models yield significant prediction errors when tested against experimental data produced with Mosalloc, indicating that the existing models might not be accurate enough for virtual memory research. The gapbs/bfs-road benchmark is missing from the Broadwell chart of Figures 5 and 6 because it is not TLB-sensitive according to our definition (its performance improves by less than 5% when backed with 1GB pages). The same workload is TLB-sensitive on older platforms like SandyBridge and Haswell, which have smaller TLBs.

Figure 7 exemplifies how we calculated the Basu model error for the gapbs/sssp-twitter workload on our SandyBridge platform. We initially executed this benchmark with 4KB pages and measured the runtime, walk cycles, and TLB misses in

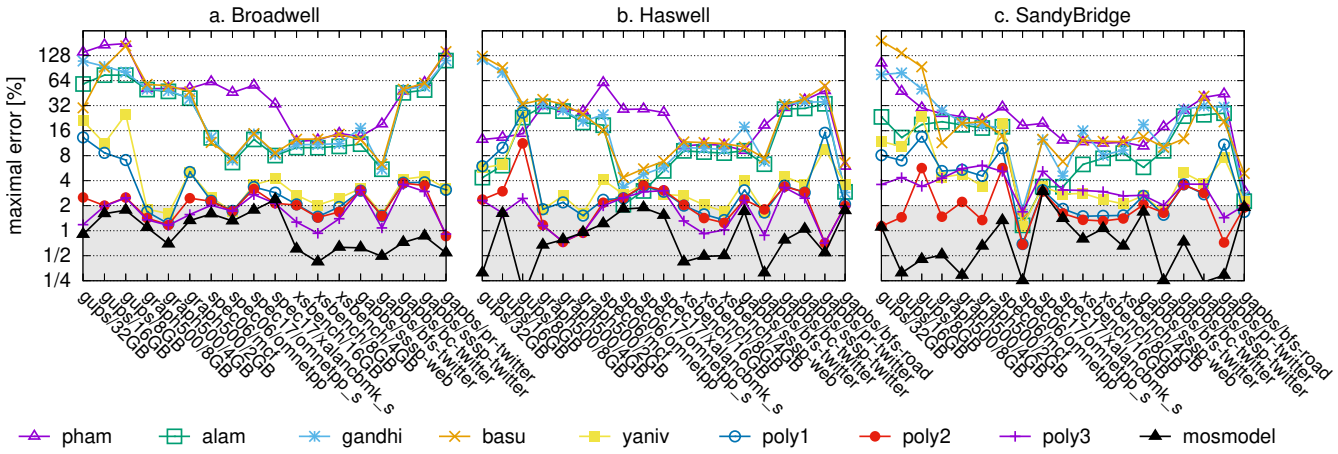

Figure 5: Per-benchmark maximal absolute prediction errors of all models; Mosmodel is typically below 2%.

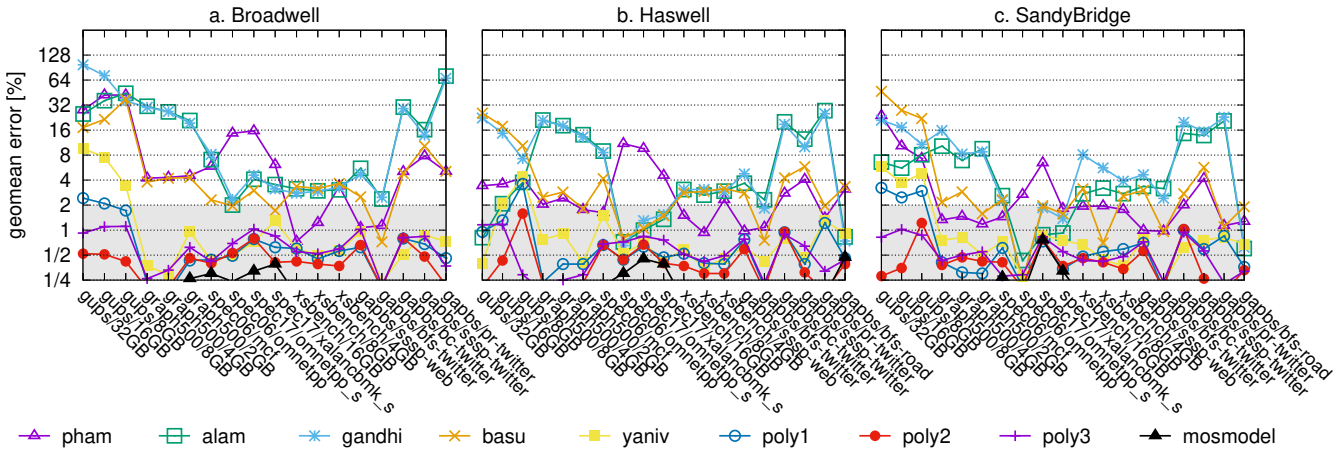

Figure 6: Per-benchmark geomean absolute prediction errors of all models; Mosmodel is typically below 0.5%.

this configuration to define the Basu model. We then used Mosalloc to measure the same workload under 54 different memory layouts that mix 2MB and 1GB pages. Basu et al. believed that their model is pessimistic, predicting runtimes that are always higher than the true values, because it does not consider the gains from eliminating L1 TLB misses that hit in L2 TLB. However, the measured data reveals that the Basu model is optimistic for this workload, predicting runtimes that are 42% lower than the true runtimes. These errors are the result of the underlying assumptions of the Basu model, as explained in Section III.

We note that the significant prediction error of the Basu model at low TLB misses configurations is alarming because these configurations are associated with (nearly) zero virtual memory overhead, which is the operational point of several recent studies. For example, the direct segment studies [8], [29] proposed hardware designs that nearly eliminate the address translation overhead. Arguably, the experimental data obtained with Mosalloc is relevant and indicative to these studies because direct segments have an affect similar to backing an application with hugepages, like is done by Mosalloc. Figure 7 suggests that these studies reported overly optimistic, unrealistically

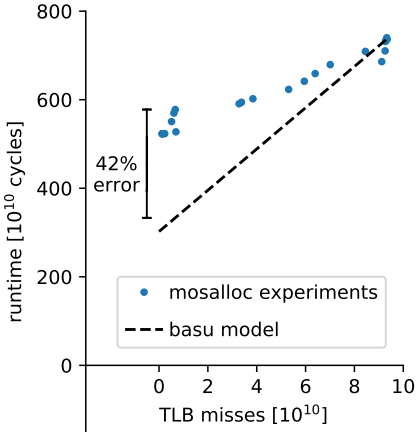

Figure 7: Runtime predicted by the linear model may be 42% lower than the real runtime for gapbs/sssp-twitter.

high performance improvements, relying on a linear model, which we now see might be inaccurate.

Several studies proposed virtual memory designs whose performance falls somewhere between 4KB and 2MB pages.

Existing linear models do not reach the maximum prediction errors (> 100%) in this region because they were calibrated to pass through the 4KB,2MB points. For example, the Basu model errors in the 4KB–2MB range lower than the maximum error achieved near the zero page walk overhead point, as demonstrated in Figure 7. These relatively-low errors might suggest that linear models are adequate for evaluating newly proposed hardware designs that operate in this range. But hardware designs that operate in this range likewise promise proportionally smaller performance gains, so those "relatively-low errors" are big enough, relatively speaking, to raise doubt regarding the validity of the evaluation results. For example, the MICRO'15 work by Pham et al. [59] operates in the 4KB–2MB range, where the Pham model indeed suffers from errors of "only" a few tens of percents. But Pham et al. reported that "on average, runtime is improved by 14%", so the error is comparable to the claimed improvement. Namely, when the performance gains are around 10%, errors of 10% are too high.

Figure 5 shows that the prediction errors on our Broadwell platform are higher compared to the SandyBridge and Haswell platforms, sometimes exceeding 100%. We analyzed these high errors and discovered that they are caused by *negative* runtime predictions of the Basu model. As mentioned in Table 4, Broadwell processors have two page table walkers that can work simultaneously. Accordingly, the hardware event C counts two "walk cycles" when these two page table walkers are active at the same cycle. The overall number of walk cycles may thus exceed the total execution cycles, as happens in the gups benchmarks. In these cases, the ideal runtime β predicted by Basu is negative.

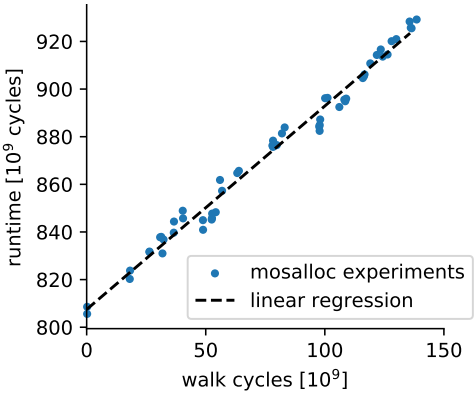

Figure 8: Linear regression describes spec06/omnetpp well.

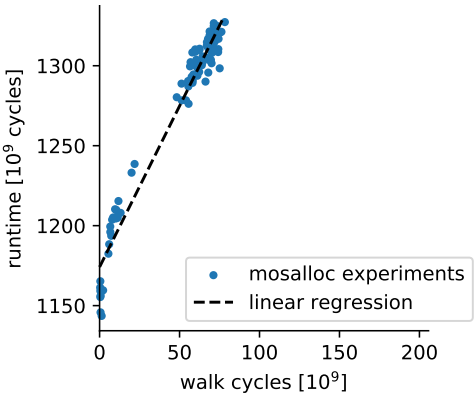

Figure 9: The slope of the spec17/xalancbmk model is > 1.

VII. Constructing an Accurate Runtime Model

Having demonstrated that the existing runtime models are inaccurate, we develop a series of more accurate models, by analyzing the muliple data points obtained with Mosalloc and by learning how real-world applications behave when their memory layout changes. We first evaluate a simple linear regression model, which achieves 26% accuracy in the worst case. Observing that the runtime is not always linear in the number of walk cycles, we evaluate higher-order polynomial models (with degree 2 and 3), which achieve up to 6% accuracy. We then utilize more inputs for the model (in addition to walk cycles) and employ Lasso regression—a statistical method to select the most relevant inputs. This new multi-input, third-order polynomial model, denoted Mosmodel, bounds the prediction errors below 3%.

A. Linear Regression Model

We first define a linear model, $R = \alpha \cdot C + \beta$, which considers the entire dataset obtained with Mosalloc (54 samples). This model differs from prior models, because they were defined by using just one or two samples. The model fits its parameters through linear regression [27], minimizing the sum of the squared errors on all measured data points and thus making it the best linear model possible for this purpose. In particular, this model is more accurate than the five linear models discussed in Section III. Figure 5 shows that the maximal prediction error of the linear regressor ("poly1") is 26% over all workloads and machines. For some workloads, e.g., spec06/omnetpp, the linear regression model is suitable, as shown in Figure 8.

Interestingly, we find that for some workloads $\alpha > 1$, which means that TLB misses might increase the runtime by more than just the table walk cycles they cause. This finding contradicts the common assumption of previous studies that TLB misses may stall the processor completely but superscalar and out-of-order processors can hide some of this overhead. Figure 9 presents the linear regression model of spec17/xalancbmk running on our Broadwell machine as an example where the slope $\alpha > 1$. While this finding is somewhat surprising at first sight, we now explain it by carefully analyzing the performance statistics of this workload.

Table 7 shows the performance counters collected from two runs of spec17/xalancbmk, one with all 4KB pages and another with all 2MB pages. The Broadwell machine L2 TLB contains 1536 shared entries for 4KB and 2MB page translations. Given that spec17/xalancbmk memory footprint is 475MB, the L2 TLB is large enough to eliminate all TLB misses for this workload when 2MB pages are used. Indeed, our measurements show that there are almost no TLB misses when the application used 2MB pages. In contrast, this workload suffers from a significant number of TLB misses when 4KB pages are

performance counter (*in **billions***)	*program* 4KB	*program* 2MB	*walker* 4KB	*walker* 2MB
runtime cycles	1320	1155		
walk cycles	76	0		
TLB misses	2	0		
L1d loads	317.1	317.1	2.0	0.0
L2 loads	64.3	64.3	1.6	0.0
L3 loads	**22.4**	**20.0**	1.0	0.0

Table 7: Runtime statistics of spec17/xalancbmk; when comparing cache loads, surprisingly, we see a difference in L3.

used. When the CPU handles these TLB misses, it inserts the page table entries to its L1/L2/L3 caches, which might cause warm application data to be evicted from the caches. Table 7 shows that the 4KB pages configuration experiences more L3 cache references than the 2MB pages configuration: $22.4 \cdot 10^9$ compared to $20 \cdot 10^9$. Some of the extra references ($2.4 \cdot 10^9$) are induced by the page walker itself ($1 \cdot 10^9$ compared to 0), and the remainder are, probably, caused by interference with the application data.

B. Polynomial Models

The linear regression model addresses the main flaw of previously-proposed linear models by accounting for more than just two experimental data points. Still, this model is not flexible enough to describe runtime accurately for all workloads, as demonstrated in Figure 3. The gups/16GB runtime does not follow a linear trend and the linear regression error may be as high as 13%. Based on this empirical observation, we suggest to favor polynomial models of degree two or three over the linear regression model. Figure 10 shows that a polynomial of degree two (poly2) is flexible enough to describe gups/16GB, predicting the measured data with a maximum error of 2%. Looking at all workloads and machines, the maximum prediction error of a third-degree polynomial model (poly3) is 6%, as shown in Figure 5. The empirical observation that application runtime sometimes behaves like a polynomial function demonstrates that the processor pipeline is able to hide the table walk cycles better when their number is lower. Conversely, when the number of walk cycles is high, the page table entries and the application working set contend for cache resources, and so the performance degrades by more than just the table walk cycles.

C. Mosmodel: Multi-Input Polynomial Models

There are multiple performance metrics associated with the virtual memory subsystem, and there is no reason to expect that one specific metric would consistently outperform the rest as a runtime predictor across all workloads. We thus suggest to extend the model input to a vector $\vec{X} = (H, M, C)$ rather than just the walk cycles C. The new multi-input, third-degree polynomial model is called Mosmodel:

$$R(H,M,C) = \beta + \alpha_0 \cdot C + \alpha_1 \cdot M + \alpha_2 \cdot H + \\ + \alpha_3 \cdot C^2 + \alpha_4 \cdot CM + \alpha_5 \cdot CH + \ldots \quad (3)$$

Models that take more inputs should, in principle, perform at least as well as models that take only walk cycles, because

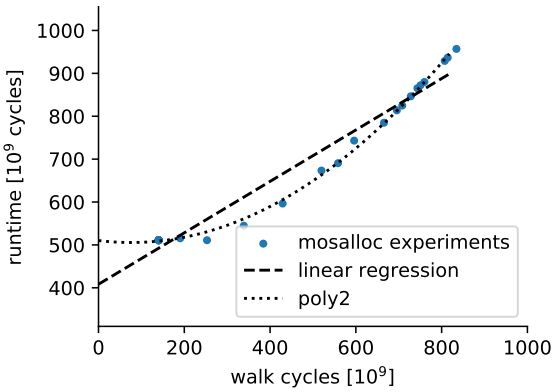

Figure 10: Linear regression is unable to accurately predict the performance of gups/16GB on SandyBridge, but the third-order polynomial is accurate enough (maximum error $< 2\%$).

the former can always ignore the additional inputs and base their prediction on the walk cycles. However, Figure 5 shows that the Mosmodel error is sometimes higher than the errors of poly1, for instance, for the spec17/xalancbmk_s benchmark. The reason is that both Mosmodel and poly1/2/3 are trained through least-squares linear regression, which minimizes the *average* squared error. Our evaluation, on the other hand, measures the *maximal* absolute error. In practice, the maximum error is correlated to the average squared error, so this issue does not change the big picture: Mosmodel performs better than poly1/2/3 and bounds the prediction error below 3% across all workloads and machines.

Mathematically speaking, extending the input vector to three variables makes the predictive model more flexible because a third-order polynomial in three variables has 20 parameters, compared to 4 parameters with a single input variable. But excessively flexible models tend to suffer from overfitting: the models will fit accurately to known data but will poorly perform when predicting the output for new, unseen inputs. To tackle the overfitting problem, we employ the Lasso regularization method for linear models [27], which basically limits the model flexibility by examining only a subset of the entire space of linear models. Lasso regression also has another advantage: it sets some of the linear coefficients to zero and effectively "selects" the relevant input variables for the model.

The newly-proposed Mosmodel raises interesting questions, e.g., which of the three inputs (H, M, C) are most useful for predicting the runtime of different workloads and why. To estimate the relative importance of these inputs, we fitted a single-variable, first-order linear regressor for C, M, and H. We then calculated the coefficient of determination, denoted R^2, which provides a measure of how well the model output (runtime in our case) are explained by the model. Table 8 reports the R^2 values of each individual input for all workloads on all machines. We see that the most useful predictors of runtime are C (the number of walk cycles) and M (the number of TLB misses). For most workloads, C and M are highly correlated, so using both of them is somewhat redundant, but for some workloads one of them is more important than the

workload	SandyBridge C	M	H	Haswell C	M	H	Broadwell C	M	H
gups/32GB	1	1	.96	1	1	.98	.99	.99	.94
gups/16GB	1	.99	.95	1	.99	.95	.99	.99	.39
gups/8GB	.99	.99	.95	.99	.98	.82	.99	.99	.72
spec06/mcf	.99	.96	.33	.95	.90	.82	.91	.94	.91
spec06/omnetpp	1	.99	.93	.97	.95	.85	.98	.97	.90
spec17/omnetpp_s	.95	.68	.90	.97	.95	.83	.95	.93	.81
spec17/xalancbmk_s	1	.99	.93	.99	.99	.91	.96	.96	.96
graph500/2GB	.96	.94	.93	.99	.99	.90	.94	.93	.89
graph500/4GB	.95	.84	.91	.99	.99	.14	.99	.99	0
graph500/8GB	.95	.72	.91	.99	.99	.76	.98	.98	.65
xsbench/4GB	.98	.98	.87	.98	.98	.01	.98	.97	.08
xsbench/8GB	.98	.97	.96	.98	.98	0	.98	.97	.02
xsbench/16GB	.99	.83	.99	.97	.96	0	.97	.96	0
gapbs/bc-twitter	.88	.79	.88	.83	.73	.68	.50	.37	.38
gapbs/bfs-road	.95	.95	.84	.90	.92	.29			
gapbs/bfs-twitter	.99	.97	.94	.94	.90	.77	.89	.86	.61
gapbs/pr-twitter	.99	.99	.85	.98	.99	.11	.99	.99	.01
gapbs/sssp-twitter	.99	.96	.76	.99	.96	0	.94	.87	.05
gapbs/sssp-web	.98	.71	.53	.96	.72	0	.94	.72	0

Table 8: The R^2 values of linear regression as a function of C (walk cycles), M (L2 TLB misses), and H (L2 TLB hits).

other. We see that H is the least valuable input because linear regression in H yields low R^2, sometimes reaching 0, which indicates that the optimal regressor is a constant function – the mean of all observations.

D. Validating Mosmodel: a Case Study

We now validate Mosmodel by applying it to predict the performance of an existing virtual memory feature: 1GB pages. In practice, computer architects should use Mosmodel to evaluate new virtual memory designs that do not exist yet, e.g., direct segments as proposed by Basu et al. [8] But since we want to validate Mosmodel, we must be able to compare its predictions against real hardware. We thus demonstrate the usefulness of Mosmodel in predicting the performance benefits from 1GB pages, which exist in all Intel microarchitectures since Westmere.

For each workload, on each machine, the training set consists the 54 Mosalloc layouts that use 4KB and 2MB pages (as described in Section VI) while the test set is the single Mosalloc layout that uses only 1GB pages. The validation procedure goes as follows: (1) measure dozens of Mosalloc layouts that use only 4KB and 2MB pages on real hardware, (2) build Mosmodel from this data, (3) measure the 1GB pages layout on real hardware (which is equivalent to simulating this configuration under a perfectly accurate partial simulator of the virtual memory subsystem), (4) apply Mosmodel to predict the runtime with 1GB pages from the "simulated" virtual memory numbers (H, M, C), (5) calculate how much the Mosmodel prediction deviates from the measured runtime of a Mosalloc layout that uses only 1GB pages, and (6) compare the Mosmodel error with the errors of past linear models.

We found that both Mosmodel and past linear models predict the 1GB pages layout accurately for most workloads and machines. For the vast majority of workloads, the runtime

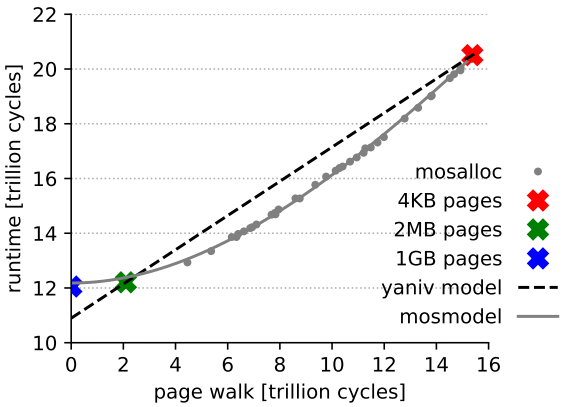

Figure 11: While the Yaniv model error is 10%, Mosmodel is accurate enough (maximum error 1%) for gapbs/pr-twitter on our SandyBridge platform.

with 1GB pages is similar to the runtime with 2MB pages, since both configurations effectively eliminate all TLB misses. Linear models that pass through the 2MB point, e.g., the Yaniv model, are thus able to predict the runtime with 1GB pages accurately. However, in several cases the existing linear models are inadequate while Mosmodel accurately predicts the runtime with 1GB pages. Figure 11 presents such example.

Evidently, the runtime of gapbs/pr-twitter on our SandyBridge platform follows a polynomial trend as a function of the page walk cycles C. It is thus not surprising the Yaniv model does not describe this workload well. Figure 11 shows that the Yaniv prediction deviates from the true runtime when 1GB pages are used by 10%. Mosmodel, on the other hand, is more flexible than the linear model, predicting the runtime accurately. Note that the spec06/mcf benchmark on SandyBridge is another example where Mosmodel outperforms preexisting models, as shown in Figure 3.

VIII. Conclusions

Runtime models are routinely used for estimating the performance of new virtual memory designs. But no previous study has ever assessed their accuracy. We validate the preexisting models using Mosalloc, a new memory allocator we develop that combines regular pages and hugepages when backing the address space of applications. Mosalloc allows us to acquire experimental data that was unavailable until now. With this data, we find that the prediction errors of preexisting models are sometimes high enough to cast doubt on previously published results, as they might deviate from real runtime by up to 25%–192%. We propose a more accurate alternative, Mosmodel, which bounds the maximal error below 3%.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd for their helpful feedback. This project received funding from VMware Research and the European Union's Horizon 2020 research and innovation programme under grant agreement No. 688386 (OPERA).

REFERENCES

[1] G. Aad, B. Abbott, J. Abdallah, A. Abdelalim, A. Abdesselam, O. Abdinov, B. Abi, M. Abolins, H. Abramowicz, H. Abreu *et al.*, "The ATLAS simulation infrastructure," *The European Physical Journal C*, vol. 70, pp. 823–874, 2010, http://link.springer.com/article/10.1140/epjc/s10052-010-1429-9.

[2] M. Agbarya, I. Yaniv, J. Gandhi, and D. Tsafrir, "mosalloc: Mosaic memory allocator," 2020, http://mosalloc.cs.technion.ac.il/. (Accessed: Sep 2020).

[3] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012, pp. 476–487, http://dx.doi.org/10.1145/2366231.2337214.

[4] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017, pp. 457–468, http://doi.org/10.1145/3079856.3080209.

[5] *AMD64 Architecture Programmer's Manual, Volume 2*, AMD, Inc., 2013, http://www.amd.com/system/files/TechDocs/24593.pdf. (Accessed: Sep 2020).

[6] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, J. Willcock, A. Korzh, and M. Zalewski, "Graph 500 benchmark specification," http://graph500.org/?page_id=12, 2017, version 2.1 (Accessed: Sep 2020).

[7] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010, pp. 48–59, http://dx.doi.org/10.1145/1815961.1815970.

[8] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013, pp. 237–248, http://dx.doi.org/10.1145/2485922.2485943.

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

[10] E. Berger, "The hoard memory allocator," http://github.com/emeryberger/Hoard, 1998, hoard git repository (Accessed: Sep 2020).

[11] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000, pp. 117–128, http://doi.org/10.1145/378993.379232.

[12] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 26–35, http://dx.doi.org/10.1145/1346281.1346286.

[13] A. Bhattacharjee, "Large-reach memory management unit caches," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 383–394, http://dx.doi.org/10.1145/2540708.2540741.

[14] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 62–73, http://dx.doi.org/10.1109/HPCA.2011.5749717.

[15] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB prefetchers for chip multiprocessors," in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 359–370, http://dx.doi.org/10.1145/1735971.1736060.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *CAN*, vol. 39, no. 2, pp. 1–7, aug 2011, http://doi.org/10.1145/2024716.2024718.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "gem5 source code – official git repository," http://gem5.googlesource.com/public/gem5, (Accessed: Sep 2020).

[18] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42, http://doi.acm.org/10.1145/3185768.3185771.

[19] L. Chen, Y. Wang, Z. Cui, Y. Huang, Y. Bao, and M. Chen, "Scattered superpage: A case for bridging the gap between superpage and page coloring," in *IEEE International Conference on Computer Design (ICCD)*, 2013, pp. 177–184, http://dx.doi.org/10.1109/ICCD.2013.6657040.

[20] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 249–261, http://doi.org/10.1109/MICRO.2007.36.

[21] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 435–448, http://doi.org/10.1145/3037697.3037704.

[22] L. documentation page, "Hugetlbpage support," http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, 2017, linux documentation page (Accessed: Sep 2020).

[23] L. documentation page, "Transparent hugepage support," http://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2017, (Accessed: Sep 2020).

[24] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 223–234.

[25] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 2010, vol. 5, http://doi.org/10.2200/S00273ED1V01Y201006CAC010.

[26] R. Felker *et al.*, "musl libc," http://musl.libc.org/, 2011, musl homepage (Accessed: Sep 2020).

[27] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. Springer Series in Statistics, 2001.

[28] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A tool to instrument x86-64 TLB misses," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 42, 2014, http://doi.acm.org/10.1145/2669594.2669599.

[29] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 178–189, http://dx.doi.org/10.1109/MICRO.2014.37.

[30] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: Exceeding the best of nested and shadow paging," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016, pp. 707–718, http://dx.doi.org/10.1109/ISCA.2016.67.

[31] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009, (Accessed: Sep 2020).

[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kauffman, 2017.

[33] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 34, no. 4, pp. 1–17, sep 2006, http://dx.doi.org/10.1145/1186736.1186737.

[34] J. L. Henning, "SPEC CPU2006 memory footprint," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 35, no. 1, pp. 84–89, mar 2007, http://doi.acm.org/10.1145/1241601.1241618.

[35] J. L. Henning, "SPEC CPU2017 benchmark descriptions," http://www.spec.org/cpu2017/Docs/index.html#benchmarks, 2018, (Accessed: Sep 2020).

[36] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, Intel Corporation, 2015, http://tinyurl.com/intel-x86-3a. (Accessed: Sep 2020).

[37] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, Intel Corporation, 2015, http://tinyurl.com/intel-x86-3b. (Accessed: Sep 2020).

[38] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, June 2016, http://tinyurl.com/intel-x86-optim. (Accessed: Sep 2020).

[39] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015, pp. 66–78, http://dx.doi.org/10.1145/2749469.2749471.

[40] D. Koester and B. Lucas, "RandomAccess – GUPS (Giga updates per second)," http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/, (Accessed: Sep 2020).

[41] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 705–721, http://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon.

[42] *brk(2) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man2/brk.2.html. (Accessed: Sep 2020).

[43] *ld.so(8) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man8/ld.so.8.html. (Accessed: Sep 2020).

[44] *malloc(3) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man3/malloc.3.html. (Accessed: Sep 2020).

[45] *mallopt(3) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man3/mallopt.3.html. (Accessed: Sep 2020).

[46] *mmap(2) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man2/mmap.2.html. (Accessed: Sep 2020).

[47] *munmap(2) man page*, Linux Programmer's Manual, http://man7.org/linux/man-pages/man2/munmap.2.html. (Accessed: Sep 2020).

[48] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *TACO*, vol. 10, no. 1, pp. 2:1–2:38, April 2013, http://dx.doi.org/10.1145/2445572.2445574.

[49] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context switch aware large TLB," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 449–462, http://doi.org/10.1145/3123939.3124549.

[50] E. Munson, "libhugetlbfs," http://github.com/libhugetlbfs/libhugetlbfs, 2015, libhugetlbfs README (Accessed: Sep 2020).

[51] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User Group Conference (CUG)*, vol. 19, pp. 45–74, 2010, http://www.richardmurphy.net/archive/cug-may2010.pdf.

[52] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 265–276, http://doi.org/10.1145/1508244.1508275.

[53] L. W. News, "Transparent huge pages in 2.6.38," http://lwn.net/Articles/423584/, 2011, (Accessed: Sep 2020).

[54] L. W. News, "Transparent huge pages in the page cache," http://lwn.net/Articles/686690, 2018, (Accessed: Sep 2020).

[55] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, 2015, http://dx.doi.org/10.1109/MM.2015.74.

[56] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *HPCA*, 2015, pp. 210–222.

[65] Wikipedia, "One in ten rule — Wikipedia, the free encyclopedia," 2020, http://en.wikipedia.org/wiki/One_in_ten_rule. (Accessed: Sep 2020).

[57] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016, pp. 217–229.

[58] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 558–567, http://dx.doi.org/10.1109/HPCA.2014.6835964.

[59] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 1–12, http://dx.doi.org/10.1145/2830772.2830773.

[60] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory TLB," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017, pp. 469–480, http://doi.org/10.1145/3079856.3080210.

[61] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 45–57, http://dx.doi.org/10.1145/605397.605403.

[62] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[63] X. Tong and A. Moshovos, "BarTLB: Barren page resistant TLB for managed runtime languages," in *IEEE International Conference on Computer Design (ICCD)*, 2014, pp. 270–277, http://dx.doi.org/10.1109/ICCD.2014.6974692.

[64] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, 2014, http://www.mcs.anl.gov/papers/P5064-0114.pdf.

[66] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," in *ACM SIGMETRICS International Conference on Measurement and Modeling (SIGMETRICS)*, 2016, pp. 337–350, http://dx.doi.org/10.1145/2896377.2901456.