Selecting Memory Layouts for Model-Based Microarchitectural Performance Evaluation

Mohammad Agbarya
a.mohammad@cs.technion.ac.il
Technion – Israel Institute of Technology
Haifa, Israel

Idan Yaniv
idanyani@cs.technion.ac.il
Technion – Israel Institute of Technology
Haifa, Israel

Dan Tsafrir
dan@cs.technion.ac.il
Technion – Israel Institute of Technology
Haifa, Israel

Abstract

Virtual memory research increasingly relies on empirical performance models that use simulated address translation metrics (x)—such as TLB miss rate and page-walk cycles—to predict benchmark runtimes (y) for new hardware designs. The accuracy of these models depends on the training data: a set of (x,y) pairs collected by executing the benchmark on real hardware with memory layouts that mix pages of different sizes. Because the number of possible layouts grows exponentially with address space size, a key challenge is to select layouts that yield informative training samples spanning a broad range of x values.

Previously proposed selectors rely on flawed heuristics that can leave large gaps in the model's input domain, necessitating a costly manual search for additional layouts. We solve this problem with Moselect, a feedback-driven layout selector that uses previously observed x values to steer selections toward uncovered regions. Moselect eliminates the manual effort, achieves good x domain coverage, converges faster than prior selectors, and significantly reduces worst-case runtime prediction error. Moreover, the higher-quality training data Moselect yields allow simpler simulators and runtime models to achieve worst-case prediction errors similar to those of their more sophisticated counterparts.

CCS Concepts: • **Computing methodologies** → **Modeling methodologies**.

Keywords: virtual memory, huge pages, address translation, TLB, page table walks, performance counters

ACM Reference Format:
Mohammad Agbarya, Idan Yaniv, and Dan Tsafrir. 2027. Selecting Memory Layouts for Model-Based Microarchitectural Performance Evaluation. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3845814.3850070



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850070

1 Introduction

Evaluating new virtual memory designs traditionally relies on cycle-level full-system simulation (e.g., gem5 [16]). Such detailed, cycle-after-cycle simulation is prohibitively expensive: up to a million times slower than native execution [5]. To reduce this cost, virtual memory research increasingly adopts *model-based evaluation* [4, 6, 7, 10, 11, 20, 27, 28, 31, 38, 45, 48, 53, 54, 60, 61]. Rather than simulating the entire system, this methodology simulates only the hardware component being studied, such as the TLB. It then uses an empirical performance model to translate the resulting simulator output x into predicted benchmark runtime y. Model-based evaluation can therefore substantially reduce evaluation cost. But its effectiveness ultimately depends on the accuracy and robustness of the runtime model, which in turn critically depends on the training data used to construct it.

Training samples can be collected by executing the benchmark on real hardware with different memory layouts containing various mixtures of 4KiB and 2MiB pages [1, 2]. For each execution, the benchmark runtime y and relevant performance counter values x (TLB misses, page-walk cycles, etc.) are recorded, yielding an (x,y) point used to construct the empirical model. Each memory layout potentially produces different address translation performance, allowing different (x,y) points to be measured on real hardware. The resulting points sample the workload's performance surface, relating virtual memory behavior to application runtime. Once constructed, the empirical model can be used to evaluate a new virtual memory design D by mapping the x values produced by simulating D to predicted runtime y. Section 2 describes this model-based evaluation methodology.

Collecting the training points poses a combinatorial selection problem. Each 2MiB memory region can be backed by one 2MiB page or multiple 4KiB pages, so the number of possible mixed-page layouts grows exponentially with application memory size. Exhaustively executing these layouts is thus impractical. Instead, a layout selector must choose a small set whose executions yield sufficiently diverse x values that cover the model's input domain. Without adequate coverage, the model must predict runtimes for inputs that are poorly represented in its training data.

Selectors The most advanced layout selection method used by prior model-based work, which we denote as the baseline selector, or `BaselineS`, does not reliably yield well-distributed

x values [2]. The reason, we find, is that it chooses new layouts while ignoring the x values obtained earlier in the selection process. Our analysis shows that x-based feedback is required because the effect of using 2MiB pages on address translation activity is surprisingly hard to predict. Layouts selected a priori may yield redundant points (similar x values) while leaving large gaps elsewhere in the x domain. Models trained on such flawed data can generalize poorly, and fixing this deficiency involves labor-intensive and time-consuming *manual* refinement of the training set [2], a burden we contend is unacceptable. Section 3 exemplifies this problem.

We introduce four new layout selector designs and compare them against BaselineS, which represents prior work. The first selector uses Intel's Precise Event-Based Sampling (PEBS), which allows us to estimate how many TLB misses are attributable to each 4KiB page. The PEBS selector, PebsS, tests whether such a static profile can successfully guide layout selection. PebsS assumes that summing the per-page estimates within a 2MiB region approximates the reduction in TLB misses expected from backing that region with a 2MiB page. It combines regions to target progressively different predicted reductions, trying to spread the resulting x values across the input domain.

Our second design is the genetic selector, GeneticS, which tests whether generic adaptive search can cope with the inaccurate x predictions that limit PebsS. GeneticS finds the largest gap between observed x values and chooses as parents the two previously evaluated layouts that produced the gap's endpoints. The child retains the 2MiB regions common to both parents and randomly inherits some of the regions on which they differ. The intuition is that mixing two layouts whose x values bracket the gap may produce an intermediate x value and thus shrink the gap.

Our third design is the Bayesian selector, BayesianS. Unlike PebsS and GeneticS, BayesianS does not explicitly target gaps in the observed x domain. Rather, it tests whether Bayesian optimization (BO), a standard approach for learning expensive black-box functions through adaptive sampling, can select a small layout set sufficient to learn the x-to-y relationship. BayesianS differs from standard BO because it cannot directly choose a desired x value. Instead, it must choose a layout, execute the workload, and only then observe the resulting x and y. To make BO usable, we encode layouts as integers using Gray code, such that adjacent values correspond to layouts differing in only one 2MiB region. We thus provide BO with a search space where nearby points tend to represent similar layouts. BO explores this space, balancing exploration with exploitation as it learns from previous executions and searches for extrema.

Our final design is the Mosaic Selector, Moselect. The "Mos" prefix refers to the "mosaic" of pages of different sizes that constitutes each memory layout. Let All_{4KiB} and All_{2MiB} denote the layouts that use only 4KiB and only 2MiB pages, respectively. Moselect treats the x values of these layouts as the endpoints of the target domain and normalizes their distance to 100%. Given a total budget of n+1 layouts, the best possible placement of the resulting x values would divide the domain into n equal gaps of size $g = 100\%/n$. Because layout effects cannot be predicted precisely, we give the selector a factor-of-two margin and define its convergence goal as achieving a maximum gap of at most $2g$.

Moselect works roughly as follows. First, it uses PEBS to construct several coarse seed layouts, executing the workload with each to produce an initial set of measured x values. Then, it repeatedly selects the largest gap between neighboring values and tries to place a new point near its middle. This is done by carefully choosing which 2MiB pages to add to the layout underlying the higher x or remove from the layout underlying the lower x. Moselect uses PEBS information as a hint to guide its choices. But it immediately executes each new layout and uses the resulting x value as feedback before deciding on the next step, adjusting accordingly. The process ends when Moselect converges or exhausts the budget. Section 4 describes the four new selectors in detail and demonstrates that Moselect converges quickly.

Model Improvements In addition to improving the (x,y) training data for runtime models through better layout selection, we improve the way these models are evaluated and constructed. Prior work cross-validates its runtime model on the dataset generated by BaselineS [2]. We apply the same procedure to every selector and call it self CV. Because the held-out points come from the same selector-generated dataset, self CV can hide overfitting to that selector's samples. Our first improvement, combined CV, evaluates each model on a common dataset generated across all selectors, providing a stricter test of generalization.

BaselineS's model, which we denote BaselineM, predicts runtime from three inputs—page-walk cycles and L2 TLB hits and misses—and it is fitted using Lasso regression. For correlated inputs, Lasso may favor some and suppress others. We find that such suppression can reduce prediction accuracy. Our second improvement, regularization, thus replaces Lasso with Ridge regression, which, empirically, handles correlated address translation inputs more robustly.

BaselineM is a third-degree polynomial over the three inputs above. The higher-quality training data from Moselect allow us to simplify the model with little loss of accuracy. Our third improvement, simplification, reduces it to a second-degree polynomial using only TLB hits and misses. This translates to reduced simulation complexity, as the simulator of the new virtual memory design (which outputs the model inputs) need not simulate page-walk cycles and thus the entire data cache hierarchy; simulating only the TLB suffices [10, 18, 60]. Section 5 explains and evaluates these improvements across selectors, showing that only Moselect consistently converges and that its training data yield lower runtime prediction error.

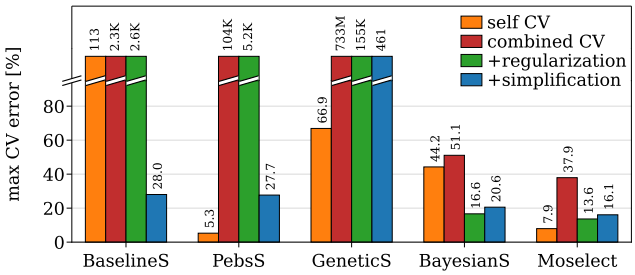


Figure 1. *Maximal absolute runtime prediction error. Comparison of our three improvements against the self CV baseline across the five selectors. Each improvement is cumulatively added to the preceding configuration. Errors are capped at 100% (y axis broken at 95%).*

Results Preview Figure 1 compares the maximal absolute runtime prediction error of the five aforementioned selectors across a set of TLB-sensitive benchmarks executed on three processor generations, highlighting the effect of successively applying the three model improvements introduced above, starting from the self CV baseline. For each selector, the figure reports four results: self CV, followed by combined CV, regularization, and simplification, with each improvement cumulatively added to the preceding configuration.

Combined CV reveals severe overfitting in models trained on the datasets produced by BaselineS, PebsS, and GeneticS, with errors becoming so large as to render the models practically unusable. In contrast, models trained on data produced by Moselect and, perhaps surprisingly, BayesianS generalize much better. Regularization then shows that Lasso's treatment of correlated inputs is a second source of error, as Ridge substantially improves Moselect and BayesianS, although it cannot rescue models trained on poorly distributed samples. Finally, simplification shows that Moselect's and BayesianS's higher-quality training data support a simpler model and simulator with little loss of accuracy. The large error reductions for some of the weaker selectors also suggest that the original three-input, third-degree BaselineM can exacerbate overfitting when the training data are poor.

Overall, BayesianS comes close to Moselect in model accuracy, but does not match it, highlighting the benefit of domain-specific selection over black-box search.

2 Model-Based Evaluation

Virtual memory is a fundamental component of computer systems, but its use comes at a cost. In x86-64, when using 4KiB pages, page tables are hierarchically organized as 4-level or 5-level radix trees that map virtual to physical addresses [8, 35]. To accelerate translation, the processor caches recently used mappings in a translation lookaside buffer (TLB). Thus, a TLB miss may trigger up to four or five memory references during the page-table walk, potentially incurring substantial overhead: some memory-intensive workloads experience slowdowns of 70% or more [27].

Virtual memory studies commonly propose new microarchitectural designs to mitigate the overhead of address translation. These studies typically use one of two evaluation methodologies. The first is a *model-less* methodology, in which the new design is evaluated using cycle-level full-system simulation [3, 13–15, 19, 22, 43, 44, 49–52, 55, 57]. While more accurate in principle, this approach is problematic because it is extremely slow, making it impractical to run benchmarks to completion. As a result, many studies use a *model-based* methodology, which simulates only the modified component(s) and estimates execution time using a mathematical model whose input is the output of the partial simulator [4, 6, 7, 10, 11, 20, 27, 28, 31, 38, 45, 48, 53, 54, 60, 61].

2.1 Overview

Figure 2a outlines the model-based evaluation workflow. To evaluate a new TLB design for a preexisting, real processor P on workload W, the methodology first builds a runtime model from measurements collected on the existing hardware and then uses that model to predict performance under the new hardware design. Specifically, the workflow consists of two phases: model construction (Steps ❶–❺) and model application (Steps ❻–❾).

During model construction, researchers ❶ select multiple memory layouts; ❷ execute W on P with these layouts using Mosalloc [1, 2], which is a memory allocator that constructs the layouts using a mixture of standard (4KiB) and huge (2MiB) pages that back W's address space; ❸ use hardware performance counters in each execution to empirically measure the runtime (y) and various virtual memory metrics (x) like TLB hits and misses; and ❹ fit a runtime model from the resulting (x,y) samples. ❺ If the model accuracy is insufficient under cross-validation, additional layouts are *manually* selected to improve the training set.

During model application (which is out of scope in this paper and is only described here to provide the full picture), researchers ❻ trace the memory references of W, typically while executing it with the default allocator; ❼ feed these references into a partial simulator that implements the proposed hardware modification (such as a new TLB design); ❽ record the simulated metric values (e.g., TLB hits and misses); and ❾ feed these metrics into the previously constructed runtime model to predict the full-system runtime on the new hardware. Compared with full-system simulation, this approach is orders of magnitude faster because only the modified component is simulated in detail [26], allowing W to run from beginning to end.

Figure 2b outlines our proposed workflow, highlighting the individual contributions with curly braces.

2.2 Mosmodel

The model-based methodology relies on a model that predicts runtime from the output of a partial simulation. Mosmodel,

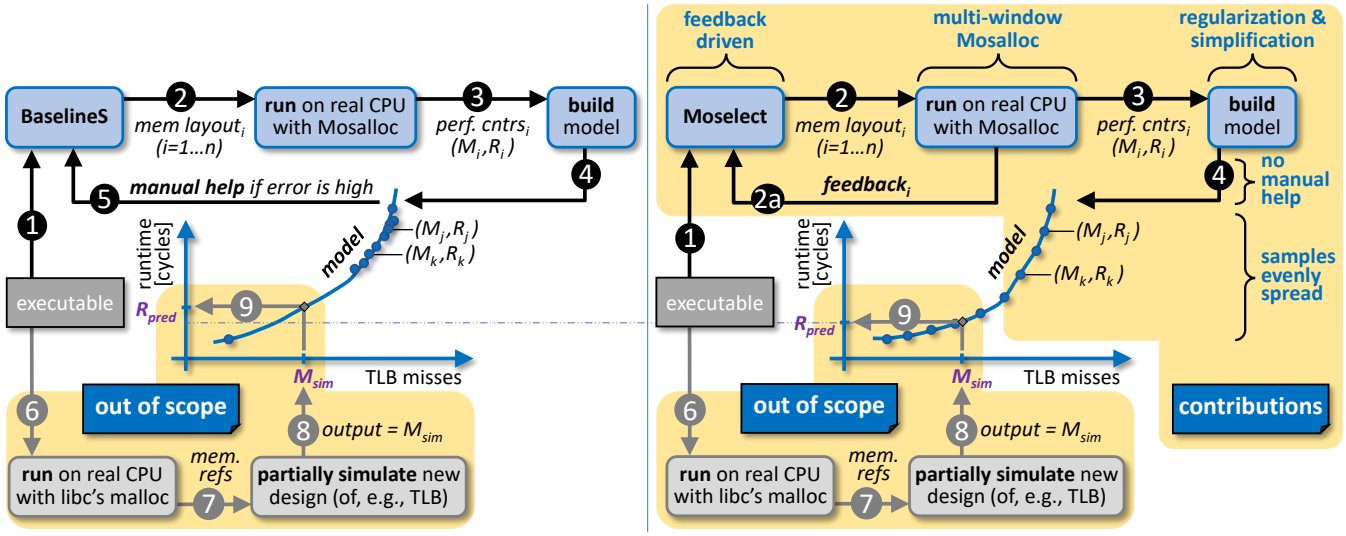


Figure 2. *(a) BaselineS may require manually selecting layouts to obtain sufficiently accurate runtime models. (b) Moselect solves this problem through feedback: immediately executing the benchmark with each newly generated layout. Text enclosed by curly braces indicates the paper's contributions.*

metric	runtime	L2 TLB hits	misses	page-walk cycles
symbol	R	H	M	C
role	output	input	input	input

Table 1. *Mosmodel metrics and the symbols that represent them.*

our prior approach, achieved a maximum prediction error below 3% when evaluated on the selector-generated samples used to fit it, with no samples held out; it improved upon earlier approaches whose maximum errors ranged from 25% to 192%. But under cross-validation, the reported maximum error of Mosmodel was 4.3%, with some benchmarks requiring manual dataset expansion to as many as 100 layouts [2]. Accuracy thus depends on the quality of both the model design and the training data.

Mosmodel is a third-degree polynomial that predicts runtime from the three virtual memory metrics specified in Table 1. Its original formulation uses Lasso regression to estimate the coefficients, providing both regularization and implicit feature selection. Unlike earlier studies, which commonly assumed a linear model based on only two measurements (All_{4KiB} and All_{2MiB}), Mosmodel leverages dozens of measurements collected under different memory layouts using Mosalloc, enabling a richer nonlinear model.

We refer to the original Lasso-based Mosmodel as BaselineM. Later, in Section 5.3, we refine the same polynomial model by replacing Lasso with Ridge regression; we refer to this variant as the *refined Mosmodel*. Subsequently, in Section 5.6, we investigate whether the higher-quality training data produced by Moselect allow the model to be simplified by reducing its polynomial degree and input set while still retaining similar prediction accuracy.

2.3 Mosalloc

As noted, accurate runtime modeling requires measurements from multiple executions of the same workload under different memory layouts (Steps ❷–❸ in Figure 2). This requirement is incompatible with standard Linux mechanisms for hugepage allocation, such as `mmap` [41], Transparent Huge Pages (THP) [21], and libhugetlbfs [46], which offer limited control (if any) over the memory layout of preexisting, unmodified applications. We speculate that this is the reason why earlier studies typically limited themselves to only using All_{4KiB} and All_{2MiB}. We developed Mosalloc to address this limitation [1, 2].

Mosalloc is a user-level library that can be dynamically preloaded when running existing binaries. It enables users to transparently back the address space of applications with a predetermined mixture of 4KiB pages and hugepages (originally using a single contiguous window of 2MiB or 1GiB pages). Mosalloc works by intercepting allocation requests and serving them from three memory pools (associated with Linux's common allocation mechanisms: `brk`, anonymous `mmap`, and file-backed `mmap`), which Mosalloc maintains and backs with physical pages corresponding to the layout definition. By making these layouts controllable, Mosalloc provides a way to obtain the diverse training samples needed for accurate runtime modeling.

2.4 Baseline Selector (BaselineS)

Because the number of possible layouts is exponential, constructing Mosmodel requires a systematic method to select which to use. The goal is not merely to generate many layouts, but to choose those producing informative samples

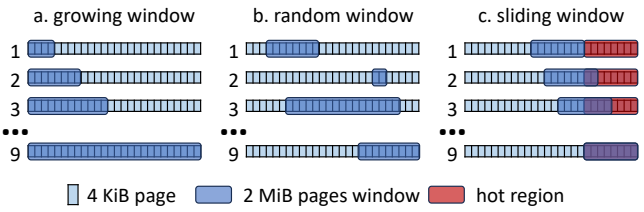


Figure 3. *BaselineS's three layout-exploration heuristics.*

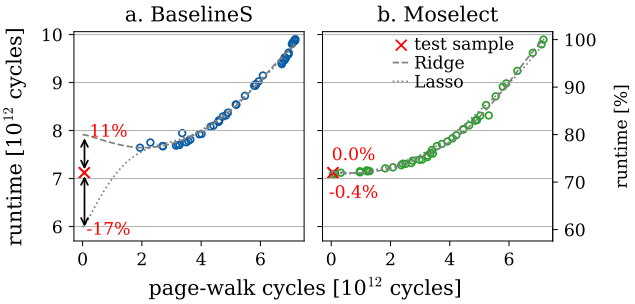


Figure 4. *Moselect's improved x domain coverage reduces the prediction error of graph500/16GiB/Skylake from 11% and -17% to 0.0% and -0.4% with Ridge and Lasso, respectively. The "test sample" point approximates the performance of the direct-segments design [11].*

across the x domain. To address this challenge, BaselineS generates 54 layouts using three static heuristics [2]. These heuristics are called "growing window," "random window," and "sliding window," and they are illustrated in Figure 3. A *window* is a contiguous region backed by 2MiB hugepages. The growing window heuristic gradually increases hugepage coverage from $\text{All}_{4\text{KiB}}$ to $\text{All}_{2\text{MiB}}$. The random window heuristic chooses a window's size and starting address at random. And the sliding window heuristic uses PEBS guidance to identify a hot region and slides a window across it. The selected layouts are rigidly restricted to a single window, and we show that they might produce many redundant samples and large gaps along the x domain.

3 The Need for a Better Selector

Accurate runtime modeling for hardware performance prediction requires training data that adequately spans the relevant operational space of the workload. In model-based virtual memory studies, the modeled runtime is inferred from measurements obtained under different memory layouts. Thus, the quality of the model depends directly on the diversity of the layouts and the corresponding measurements. Insufficient diversity introduces redundancy, weakens generalization, and can lead to inaccurate predictions even when the regression model itself is expressive [29].

Figure 4a illustrates this point with graph500/16GiB running on Skylake. The layouts selected by BaselineS leave a large uncovered region around the held-out point at the left. We claim that this point approximates the performance of the "direct-segments" design [11], which prevents most TLB misses. We corroborate our claim by executing the workload with Mosalloc using the $\text{All}_{1\text{GiB}}$ layout, which backs the 16 GiB address space using 16 × 1GiB hugepages. Because Skylake's L2 TLB has 16 entries for such pages [37], this setup nearly eliminates all page walks. The result, denoted "test sample" in the figure, constitutes the only experiment in this paper that uses 1GiB pages. Although Moselect is limited to using 4KiB and 2MiB pages only, it can nevertheless largely eliminate the error by densely covering the near-zero neighborhood of the held-out point with 2MiB hugepages.

BaselineM, which uses Lasso regression, predicts the held-out point with a -17% error when trained on BaselineS layouts, significantly overestimating the performance gain. Refining BaselineM by using Ridge instead of Lasso (which can be unstable with correlated inputs; see Section 5.3) reduces the absolute runtime prediction error to 11%, although the model now underestimates the performance gain.

In contrast, when trained on layouts generated by Moselect, the Lasso and Ridge models yield errors of only -0.4% and 0.0%, respectively, as shown in Figure 4b. Thus, while the Ridge refinement improves the model's accuracy, regularization alone cannot compensate for poorly distributed training data. We note in passing that the curve trends at the left of Figure 4a are affected by data not shown. Specifically, Mosmodel predicts runtime from three inputs (H, M, C), but the figure shows only a two-dimensional projection over walk cycles (C), so the effect of the other inputs is not visible. The key point, however, still stands: wide gaps within the training data can make the model less reliable in those uncovered regions.

We use a coverage metric to quantify how well selectors cover the x domain. For each benchmark, we normalize the interval bounded by the measured x values of $\text{All}_{4\text{KiB}}$ and $\text{All}_{2\text{MiB}}$ to span 100%. Values are considered "covered" if they lie within 2 percentage points of a measured x value obtained by running the benchmark with a memory layout generated by the selector. We define the "uncovered range" as the total length of the uncovered intervals, expressed as a percentage of the normalized domain. Figure 5 illustrates this metric.

Using this metric for the above workload, Figure 6 compares the coverage of Moselect and BaselineS. Moselect is able to achieve full coverage with its (50-layout) budget. BaselineS, in contrast, reduces the uncovered range more slowly, consuming its entire (54-layout) budget while still leaving 33% of the x domain uncovered. As we have seen, this remaining gap is large enough to materially degrade the accuracy of the model (Figure 4). Although `graph500` contains hot regions, BaselineS's sliding-window heuristic—which targets the hottest one—fails to provide full coverage. This failure suggests that the situation can be worse for workloads with weaker, more diffuse locality (such as GUPS [39]), for which sliding-window exploration is less informative.

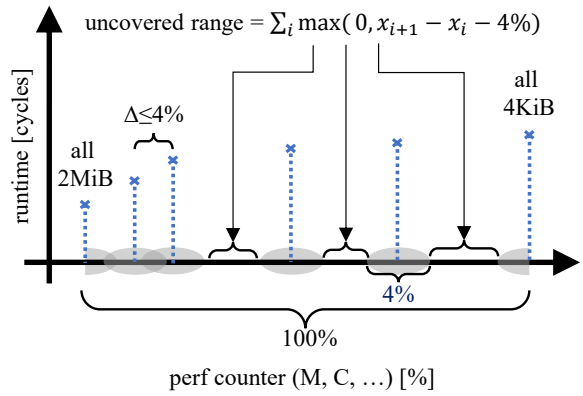


Figure 5. *For a given benchmark and selector, the uncovered range is the portion of the normalized x domain lying more than 2 percentage points from every measured sample. (Here, x_i denotes the measured normalized values in increasing order.)*

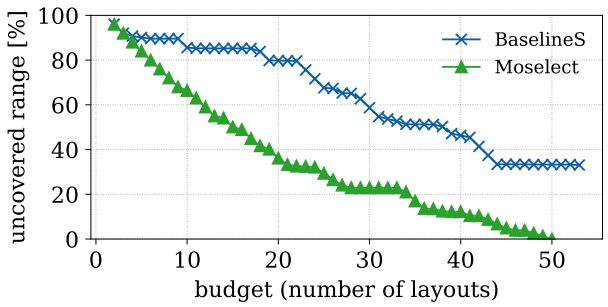


Figure 6. *Moselect achieves full coverage, while BaselineS leaves 33% of the range uncovered for graph500/16GiB/Skylake.*

In the past, when BaselineS left wide coverage gaps for benchmarks such as graph500, we occasionally had to resort to manually finding memory layouts that generate x values within these gaps [2]. This process required expert knowledge of the Mosmodel toolchain and was labor-intensive, sometimes necessitating days of work for a single workload. While doing this task, we sometimes had to work around a BaselineS drawback rooted in its underlying allocator, Mosalloc [1], which supports only a single contiguous hugepage window. This limitation needlessly restricts the memory layouts that can be explored, potentially preventing the selector from finding useful, more diverse points. The limitation is particularly restrictive for workloads with multiple, non-contiguous hot regions (shown later in Figure 7).

In light of the above, our primary goal in this paper is to automate the process of finding layouts that uniformly cover the entire x domain, thereby allowing non-expert selector users to quickly generate high-quality training data. (An immediate first step is to extend Mosalloc to support backing multiple non-contiguous regions with hugepages.) Our second goal is to answer the question: must layout selection be treated purely as a coverage-driven task, or can black-box

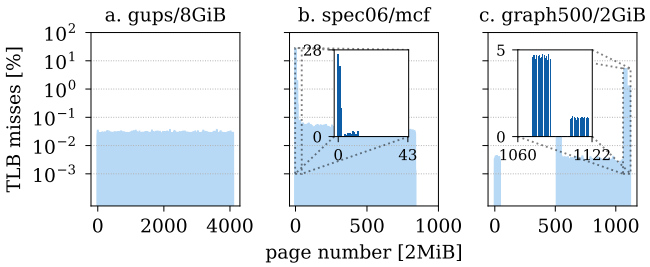


Figure 7. *Normalized PEBS histograms of TLB misses at 2MiB granularity for three Haswell workloads, showing **(a)** no hot regions, **(b)** a single hot region, and **(c)** multiple hot regions. Insets in **(b)** and **(c)** zoom in on the hot regions.*

methods such as Bayesian optimization be adapted to provide an effective alternative? The next section explores the selector design space to this end.

4 Newly Proposed Selectors

When using the model-based methodology of Figure 2, the layout selector generates the training data used to build the model. Our objective is to choose layouts whose measurements are distributed as evenly as possible in the x domain, while keeping the execution cost comparable to that of BaselineS. For $n+1$ layouts, the ideal spacing yields n equal gaps of $g = 100\%/n$. Since layout effects are not exactly predictable, we define convergence as a maximum gap of at most $2g$. With a budget of 50 layouts, $2g$ is approximately 4%; we therefore use the slightly stricter target $\max(\Delta x) \leq 4\%$ throughout our evaluation. In practice, a selector cannot directly choose an x value: it chooses a memory layout whose x value becomes known only after executing the workload. The 50-layout budget therefore allows the selector to recover from layouts that do not land at their intended positions without exceeding the measurement cost of BaselineS.

Walk Cycles as the x Domain Mosmodel is a third-degree polynomial over three inputs, as described in Table 1. Although TLB misses (M) and walk cycles (C) strongly affect runtime, we prioritize C during selection because a TLB miss does not have a fixed cost. A page walk may access different levels of the memory hierarchy depending on page-table locality and cache residency [37]. Consequently, the same number of TLB misses can incur different numbers of walk cycles and ultimately different runtimes. We therefore use walk cycles as the x axis whose coverage the selector seeks to improve.

Multi-Window Mosalloc Before describing the selectors, we first extend the layout-construction mechanism available to them. The original single-window Mosalloc supports only a single contiguous window of 2MiB pages, restricting which combinations of memory regions can be evaluated. We therefore introduce the multi-window Mosalloc, which supports

multiple non-contiguous windows of 4KiB, 2MiB, or 1GiB pages at arbitrary aligned locations. This flexibility allows selectors to manipulate individual regions independently and serves as the execution substrate for the layouts studied below.

PEBS Profile Precise profiling of walk cycles at page granularity is not available, so we use Intel PEBS measurements of retired TLB miss events to obtain region-level guidance. Modern processors expose such events through the performance monitoring unit (PMU) [36], which can be accessed through Linux *perf* [40, 42]. Intel's precise event-based sampling (PEBS) additionally associates sampled events with memory addresses. For each workload, we generate a one-time profile by executing it with $\text{All}_{4\text{KiB}}$ and sampling retired L2 TLB misses through MEM_INST_RETIRED:STLB_MISS_LOADS and MEM_INST_RETIRED:STLB_MISS_STORES.

We aggregate the sampled addresses into bins matching 2MiB regions. Each bin therefore corresponds to a candidate hugepage, and its normalized pebsWeight estimates the fraction of sampled TLB misses attributable to that region:

$$pebsWeight(hugepage_i) = \frac{hist[i]}{\sum_k hist[k]},$$
$$pebsWeight(L) = \sum_{j \in L} pebsWeight(hugepage_j),$$

where L is the set of regions backed by hugepages in a layout. Thus, a region with a large pebsWeight is considered *hot*, whereas a region with a small weight is considered *cold*.

Figure 7 illustrates these patterns for three Haswell workloads: (a) gups/8GiB has no dominant hot region, (b) spec 06/mcf concentrates many misses in a small contiguous area, and (c) graph500/2GiB has multiple disjoint hot regions. These patterns suggest that PEBS can provide useful region-level guidance for layout construction. However, because the profile is collected only once under $\text{All}_{4\text{KiB}}$, it only approximates the effect of changing a region's page size.

We next explore progressively more adaptive selection approaches. We first ask whether static PEBS guidance alone is sufficient with PebsS. When it is not, we use GeneticS to test whether generic adaptive selection can close the observed gaps. We then step outside explicit coverage-driven selection and evaluate BayesianS, a black-box optimization approach. Finally, drawing on the limitations exposed by these alternatives, we introduce Moselect, which combines PEBS-guided layout construction with feedback from measured executions.

4.1 PEBS Selector (PebsS)

Our simplest approach, PebsS, uses the above PEBS profile for static layout selection. PebsS sorts the 2MiB regions by descending pebsWeight, greedily adding regions until a desired cumulative weight is reached. Specifically, it selects layouts with pebsWeight from 0% to 100% in 2% increments, amounting to a total of 51 layouts (one more layout than the budget allocated to the rest of our newly proposed selectors, as 100%/2%=50 intervals are defined by 51 points).

The limitation is that PebsS treats the one-time PEBS profile as an accurate predictor of the effects of later layouts. In practice, changing one region to a hugepage alters TLB behavior and can also change the effect of other regions. Region effects are therefore neither perfectly predictable nor additive. As shown later in Table 2, the actual benefit can differ substantially from the static estimate. Consequently, layouts that appear evenly spaced according to pebsWeight need not be evenly spaced after execution. This motivates adding feedback and selecting layouts dynamically.

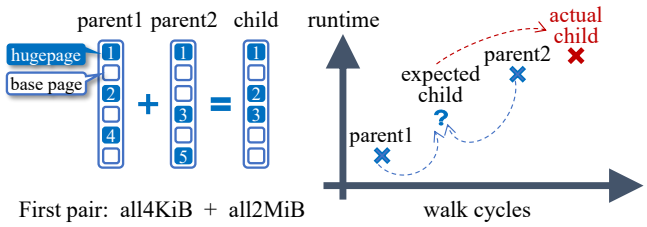


Figure 8. *GeneticS combines two parent layouts to generate a child layout, which is expected to fall between the parents but may instead fall far outside their range.*

4.2 Genetic Selector (GeneticS)

A natural next step is to use the measurements obtained during selection as feedback. We therefore developed GeneticS, which treats layout construction as a subset-selection problem. It starts from the two extreme layouts, $\text{All}_{4\text{KiB}}$ and $\text{All}_{2\text{MiB}}$. After executing them, it identifies the largest gap between measured x values and selects the two layouts that bound this gap as parents. It then creates a child layout intended to fall between them.

As illustrated in Figure 8, the child keeps all hugepages common to both parents and randomly inherits some of the regions on which they differ. If region effects were approximately additive, combining two layouts whose measured values bracket a gap would tend to produce an intermediate point and shrink that gap.

In practice, this assumption is unreliable. A child can fall far outside the interval between its parents because changing a hot region may have a much larger or smaller effect than changing several colder regions. The targeted gap then remains open, hampering the progress of GeneticS. In some workloads it therefore fails to achieve the coverage target. This exposes an important distinction: measured feedback tells the selector *where* an additional sample is needed, but the genetic crossover provides little information about *how* to construct a layout that will place the sample there. Rather than immediately introducing a more specialized heuristic, we next ask whether a general black-box optimizer can navigate the layout space directly.

4.3 Bayesian Selector (BayesianS)

The limitation of GeneticS motivates a fundamentally different approach. Instead of identifying an uncovered interval and explicitly constructing a layout inside it, we ask whether a general black-box optimizer can learn promising layouts from previous executions. This motivates Bayesian Optimization (BO), which is designed for expensive black-box functions. BO maintains a probabilistic surrogate, typically a Gaussian process (GP), and uses an acquisition function to balance exploration of uncertain regions against exploitation of promising ones [24]. We adapt this approach to memory layouts in BayesianS.

We formulate the BO objective as

$$L^* = \arg\min_L M(L),$$

where L is a memory layout and $M(L)$ is its measured L2 TLB miss count. Thus, the GP surrogate learned by BayesianS approximates the layout-to-miss relationship, $L \mapsto M(L)$.

We use BO here as a sampling policy rather than as an optimizer whose optimum we need. Over the 4KiB/2MiB layout space, $M(L)$ is minimized by $\text{All}_{2\text{MiB}}$. What we test is therefore not whether BO finds that minimum, but which layouts it visits while learning $L \mapsto M(L)$ well enough to search for one.

Evaluating $M(L)$ requires executing the workload under L on real hardware. Our implementation uses `gp_minimize` from `scikit-optimize` with Expected Improvement (EI) as the acquisition function. EI favors candidates that may improve the best objective observed so far while accounting for uncertainty in the GP prediction. Importantly, BayesianS does not use walk cycles when choosing layouts: walk cycles (C) define the x axis on which we evaluate selector coverage, whereas BayesianS optimizes the measured L2 TLB miss count (M).

Applying conventional GP-based BO directly to memory layouts is challenging because the search space is discrete and exponentially large. A workload spanning N 2MiB regions admits 2^N layouts, naturally represented by an N-bit vector in which each bit specifies whether one region uses a hugepage. Exposing every bit as a separate BO dimension, however, would create hundreds or thousands of dimensions. We therefore transform the layout bit vector with reflected Gray coding [30] and pack consecutive Gray-code bits into integer-valued BO dimensions. This representation mitigates the discontinuities of a direct binary-to-integer encoding and makes nearby coordinates more likely to represent similar layouts. Every BO proposal is decoded back into a page layout before execution.

We limit the representation to at most 20 BO dimensions, each containing up to 63 Gray-code bits so that its value remains within the signed 64-bit range. This representation supports up to 20×63 = 1,260 logical regions (each backed by either base or huge pages). For larger footprints, BayesianS groups adjacent 2MiB regions into a single logical decision, increasing the grouping factor just enough to keep the complete workload within 20 dimensions. This approach trades spatial granularity for bounded BO dimensionality. Bounding the number of logical regions means that workloads whose footprints exceed 1,260 2MiB regions are represented at a coarser region granularity than by the other selectors.

BayesianS spends its 50-layout budget as follows. First, it evaluates $\text{All}_{4\text{KiB}}$ and $\text{All}_{2\text{MiB}}$, supplying these observations to `gp_minimize`. The latter performs its default procedure, drawing 10 unguided points and then using EI as the acquisition function for the remaining 38 GP-guided evaluations. For every proposed candidate, BayesianS decodes the BO coordinates, executes the resulting layout, measures its L2 TLB misses, and adds the observation to the BO dataset. The GP is then updated before selecting the next candidate. We do not apply a separate posterior-convergence test; selection ends when the evaluation budget is exhausted.

We do not tune BO separately for individual workloads. Apart from the layout representation, the 20-dimensional limit, EI, and the evaluation budget, we retain the standard GP and optimizer behavior of `scikit-optimize`, including its internal hyperparameter fitting. Thus, BayesianS does not require additional workload-specific BO parameter tuning.

The layouts selected by BayesianS should therefore be interpreted in terms of its L2 TLB miss objective rather than our walk-cycle coverage objective. Regions that strongly affect TLB misses can influence the GP indirectly through the measured objective, but BO has no explicit knowledge of region-level hotspots or walk cycles during adaptive selection. Our implementation also supports alternative initialization strategies, including PEBS-guided initialization that can bias the initial layouts toward TLB hot regions. We leave systematic evaluation of these alternatives, including their effects on convergence and selected layouts, for future work.

This different objective also limits BayesianS as a solution to our coverage problem. Because it selects layouts to optimize L2 TLB misses rather than to fill uncovered walk-cycle intervals, several selected layouts can produce nearby x values while leaving large gaps elsewhere. As Section 5.2 shows, BayesianS does not consistently satisfy our target $\max(\Delta x) \leq 4\%$. This does not mean that BO fails at its own optimization objective; indeed, Section 5.4 shows that its layouts can still produce surprisingly accurate runtime models. It does show, however, that black-box optimization does not directly solve our explicit coverage problem. We therefore return to coverage-driven selection and design a domain-specific approach that combines region-level guidance with execution feedback.

4.4 Moselect

The preceding approaches reveal the ingredients needed for reliable coverage-driven selection. PebsS shows that PEBS provides useful region-level guidance, but static predictions

layout	*layout pages*				*L2 TLB miss reduction*	
number	P_1	P_2	P_3	P_{4+}	*expected*	*measured*
1	4KiB	4KiB	4KiB	4KiB	0%	0%
2	2MiB	4KiB	4KiB	4KiB	25%	60%
3	4KiB	2MiB	4KiB	4KiB	22%	64%
4	4KiB	4KiB	2MiB	4KiB	8%	32%
5	2MiB	2MiB	4KiB	4KiB	47%	76%
6	2MiB	4KiB	2MiB	4KiB	34%	72%
7	4KiB	2MiB	2MiB	4KiB	30%	73%
8	2MiB	2MiB	2MiB	4KiB	56%	78%
9	2MiB	2MiB	2MiB	2MiB	100%	100%

Table 2. *Expected and measured L2 TLB miss reductions for spec 06/mcf on Haswell. Layouts 1–8 cover all 2^3 page-size combinations for the first three 2MiB regions, P_1–P_3 (0–6 MiB), which are also the three hottest regions. P_{4+} denotes all remaining regions. Layout 1 is All_{4KiB} and layout 9 is All_{2MiB}. Highlighted entries indicate regions backed by hugepages. All reductions are relative to All_{4KiB}.*

alone are insufficient. GeneticS shows that measured feedback identifies gaps, but unguided layout modifications cannot reliably place new samples inside them. BayesianS shows that a general optimizer can find useful layouts, but it does not explicitly enforce coverage of the walk-cycle domain. These observations motivate Moselect: a domain-specific selector that uses PEBS to guide layout construction, and measured executions to correct its predictions and close the remaining gaps in the x domain.

Our first feedback-driven design started from only All_{4KiB} and All_{2MiB} and iteratively tried to construct intermediate layouts. This approach still failed to close some gaps reliably. Adding or removing a hot 2MiB region could change TLB misses and walk cycles far more, or far less, than its pebsWeight suggested, causing a layout intended for one part of the x domain to land elsewhere. Moselect therefore uses a hybrid two-phase design: it first creates a diverse set of PEBS-guided *anchor layouts*, and then uses execution feedback to refine the measured gaps between those anchors.

Why Feedback Is Necessary The difficulty stems from the context-dependent effect of hugepages. The pebsWeight is derived from one PEBS profile under All_{4KiB} and estimates the fraction of sampled L2 TLB misses associated with each 2MiB region. If region effects were independent, the weights of selected regions could be added to predict a layout's effect. In practice, changing one region can alter the benefit obtained from others, making this assumption inaccurate.

Table 2 illustrates three limitations of using PEBS weights as predictions. First, the predicted reduction can differ substantially from measurement: layout 2, for example, is expected to reduce TLB misses by 25% but achieves 60%, a 35-percentage-point difference. Second, layout effects are strongly nonadditive. Merging layouts 2 and 3 promotes both P_1 and P_2; their PEBS weights predict a 47% reduction, yet the merged layout (layout 5) achieves 76%. Moreover, the two layouts individually achieve reductions of 60% and 64%, whose sum exceeds 100%, demonstrating that their measured benefits overlap and cannot be treated as independent contributions. Third, the predicted direction can reverse: layout 2 is expected to outperform layout 3 (25% versus 22%), but measurement places layout 3 ahead (64% versus 60%). We call this an *ordering inversion*.

machine	*absolute error (percentage points)*			*flips*
	maximum	*mean*	*median*	
Skylake	48.21	6.35	3.81	1.9%
Broadwell	43.57	6.50	4.02	2.7%
Haswell	47.65	7.16	2.93	3.2%

Table 3. *PEBS prediction errors across all workloads. Flips show the fraction of consecutive layout pairs for which PEBS predicts that one layout will incur fewer L2 TLB misses than the other, but measurements show the opposite.*

These effects generalize across workloads and platforms. Table 3 shows that PEBS predictions have mean absolute errors of 6.35–7.16 percentage points across the three machines, while maximum absolute errors reach 43.57–48.21 points. Across successive layouts, 1.9%–3.2% of predicted directions are inverted by measurement. Thus, PEBS provides useful region-level guidance, but can occasionally place a generated layout on the opposite side from expected, motivating Moselect's execution feedback.

Static Initialization The behavior above motivates both execution feedback and the design of our initial anchors. Starting the feedback-driven search from only the two endpoint layouts provides too little information about context-dependent region effects. Moreover, concentrating the hottest regions in the same layouts would repeatedly couple their effects, making it difficult to observe how different combinations behave. Moselect therefore begins with a small, diverse set of PEBS-guided anchors that distribute high-weight regions across different groups. It sorts the profiled 2MiB regions by decreasing pebsWeight and processes them one at a time, greedily assigning each region to one of three disjoint groups with target weights of approximately 14%, 28%, and 56%. Processing the hottest regions first spreads large-weight regions across different groups rather than concentrating them in a single group; lower-weight regions subsequently fill the remaining capacity. The resulting groups therefore contain different mixtures of hot and cold regions.

Moselect enumerates all 2^3 combinations of these groups. Because each target is roughly double the previous one, their subset sums are spread nearly evenly: 0%, 14%, 28%, 42%, 56%, 70%, 84%, and 98%. All_{2MiB} provides a ninth anchor at 100%. Table 4 shows the anchors sorted by expected pebsWeight. If PEBS is a reasonable first-order guide, these anchors provide coarse domain coverage before dynamic refinement begins.

The diversity of these anchors is as important as their expected spacing. By distributing hot regions across separate

layout	*subset*	*pebs*	*layout*	*subset*	*pebs*
1	$[\emptyset]$	0%	**6**	$[S_1, S_3]$	70%
2	$[S_1]$	14%	**7**	$[S_2, S_3]$	84%
3	$[S_2]$	28%	**8**	$[S_1, S_2, S_3]$	98%
4	$[S_1, S_2]$	42%	**9**	All_{2MiB}	100%
5	$[S_3]$	56%			

Table 4. *Moselect's nine PEBS-guided anchor layouts, ordered by increasing expected pebsWeight.*

groups and combining the groups in different ways, the static phase avoids a nested sequence in which each layout simply extends the previous one. After execution, the anchors are reordered by their measured x values, which may differ from their PEBS-predicted order. Neighboring measured anchors therefore tend to offer different mixtures of hot and cold regions that the dynamic phase can add or remove when constructing an intermediate layout. Thus, the static phase provides both coarse initial coverage and a richer set of region-level choices for later gap closing.

Many Moselect layouts place hugepages in multiple non-contiguous regions and therefore require the multi-window Mosalloc for execution; the single-window Mosalloc supports only layouts with a single contiguous hugepage region.

Moselect executes the nine anchors and measures their walk cycles, then sorts them by their positions in the normalized x domain. Each pair of adjacent anchors defines an interval for refinement. The remaining layout budget is allocated in proportion to the measured interval sizes, giving larger gaps more opportunities for refinement.

Feedback-Driven Refinement Within each interval, Moselect uses the measured layouts to determine where another point is needed. For a remaining gap, it targets an intermediate measured position, typically near the gap midpoint. However, the selector cannot directly request a desired walk-cycle value; it can only modify a memory layout, execute the workload, and observe the resulting x. Moselect therefore uses PEBS to estimate which page changes may move an existing layout toward the target.

Let L_R and L_L denote the two layouts bounding the current interval, represented as sets of regions backed by hugepages, and let U denote all candidate 2MiB regions. Moselect forms

$$\alpha = L_R \setminus L_L, \qquad \beta = L_L \setminus L_R, \qquad \gamma = U \setminus (L_R \cup L_L).$$

Here, α and β contain regions selected by only one endpoint, whereas γ contains regions selected by neither. These sets give Moselect controlled pools of pages to add or remove, instead of changing the layout arbitrarily.

Moselect first tries PEBS-guided page additions, preferring eligible regions outside the two endpoints and expanding the candidate set when needed. It can also remove regions from the appropriate endpoint-specific set. Within these sets, regions with low pebsWeight allow fine-grained adjustments, whereas heavier pages make larger changes. The desired cumulative PEBS weight is estimated from the target position and from the relationship observed in previous executions. The candidate is then executed, and its actual walk-cycle value becomes feedback for the next decision.

If a new sample lands near the target, it reduces the gap and becomes part of subsequent refinement. If it moves differently than expected, Moselect uses the discrepancy between its predicted and measured positions to adjust the next PEBS target. When one strategy makes insufficient progress, the selector can switch between adding and removing pages, reverse the PEBS-weight ordering, or mix the layouts bounding the largest remaining gap. Thus, PEBS proposes promising page changes, but measured execution feedback determines how the search proceeds.

The refinement continues until $\max(\Delta x) \leq 4\%$ or the global layout budget is exhausted. Intervals that already meet the target receive no further evaluations, and wider intervals receive larger budgets. If an interval consumes its allocation without closing its gaps, the remaining budget can be redirected toward the largest unresolved gap. Conceptually, Moselect repeatedly performs the same closed loop: identify a gap, construct a PEBS-guided layout, execute it, measure where it actually lands, and use that observation to guide the next layout. This combination of structured initialization, region-level guidance, and online correction distinguishes Moselect from the preceding approaches.

Discussion and Generalization Moselect addresses the training-data problem rather than changing the runtime model itself. Its feedback-based selection determines *which* layouts should be measured, while the multi-window Mosalloc provides the flexible allocation mechanism needed to execute them. The resulting better-distributed training points enable two additional improvements evaluated in Section 5. First, replacing Lasso in BaselineM with Ridge improves robustness to the correlated inputs (H, M, C). Second, the higher-quality Moselect points allow us to reduce model and simulator complexity while retaining similar accuracy. These contributions are complementary: Moselect and the multi-window Mosalloc improve which measurements are collected, whereas Ridge and model simplification improve how those measurements are used.

Moselect currently selects between 4KiB and 2MiB pages only. A natural extension is to apply the heuristic hierarchically. As a first stage, PEBS information could be aggregated at 1GiB granularity, and the same static-plus-feedback procedure could determine which aligned regions should use 1GiB pages, while the remaining memory is backed by 2MiB pages. The selector could then be applied recursively within regions not promoted to 1GiB pages to choose between 2MiB and 4KiB pages. The multi-window Mosalloc already provides the multi-window page-size control needed to realize such layouts; designing and evaluating this hierarchical selector is left for future work.

generation	*processor (cores × sockets)*	*main memory*
Haswell	Xeon E7-4830 v3 2.1 GHz (12×2)	128 GiB (1.6 GHz)
Broadwell	Xeon E7-8890 v4 2.2 GHz (24×4)	512 GiB (2.4 GHz)
Skylake	Xeon Silver 4116 2.1 GHz (12×2)	128 GiB (2.4 GHz)

Table 5. *Evaluation platforms used throughout the study.*

suite	*refs*	*benchmark*	*footprint*	*N*
SPEC CPU2006	[32, 33]	429.mcf	1.7 GiB	2
		471.omnetpp	196 MiB	
SPEC CPU2017	[17, 34]	620.omnetpp_s	264 MiB	2
		623.xalancbmk_s	506 MiB	
GAP (multi-threaded kernels on real-world graphs)	[12]	BC/twitter	13.2 GiB	6
		BFS/road	1.0 GiB	
		BFS/twitter	12.4 GiB	
		PR/twitter	12.3 GiB	
		SSSP/twitter	23.5 GiB	
		SSSP/web	30.0 GiB	
Graph500	[9, 47]	compress and BFS	2/4/16 GiB	3
XSBench	[58]	Monte Carlo sim.	4/8/16 GiB	3
GUPS	[39]	random reads	8/16/32 GiB	3
total				19

Table 6. *TLB-sensitive workloads used to evaluate selector coverage and model accuracy. Footprints measured during execution. N counts per-suite workloads, including footprint variants.*

5 Model Improvements and Evaluation

We first describe our experimental methodology and evaluation metrics, then compare the selectors' coverage of the x domain. We next evaluate runtime-model accuracy and the effect of replacing Lasso with Ridge, before examining piecewise linearity, model simplification, and whether self CV can reliably guide model selection.

5.1 Methodology

Benchmarks and Platforms We evaluate Moselect on three Intel microarchitectures: Haswell, Broadwell, and Skylake. Table 5 summarizes the systems. Haswell and Broadwell are retained for comparability with the prior study [2], whereas Skylake is added to test whether the conclusions extend to a newer Intel generation. We exclude Sandy Bridge because it lacks the PEBS support required by our profiling methodology. Hyper-threading is disabled, the operating system is configured for maximum performance, and all machines run Ubuntu 20.04.2 LTS with Linux kernel 5.4.0. We do not evaluate AMD or ARM platforms because our methodology requires precise walk-cycle measurements, which are not available in the required form.

To stress the address translation subsystem, we use the 19 TLB-sensitive workloads (whose runtime changed by at least 5% when backed with 1 GiB pages) from the prior study that introduced BaselineS [2], as summarized in Table 6. For each selector and workload, we repeat every experiment four times and discard outliers. To minimize NUMA effects, each run is pinned to the cores and memory of a single socket.

Layout Budget The BaselineS budget is hardcoded to 54 layouts (Section 2.4). As noted, PebsS is allocated 51 layouts (Section 4.1), and the rest of our selectors—GeneticS, BayesianS, and Moselect—are allocated 50 layouts (Section 4). In principle, we could have equalized the budget by allocating 54 layouts to all selectors. But it is doubtful that doing so would have changed the results significantly, as the budgets differ by at most four layouts. More importantly, Moselect receives the same budget as GeneticS and BayesianS and a smaller budget than BaselineS and PebsS, giving it no budget advantage over the other selectors.

Metrics We evaluate selector quality through the uncovered range and its convergence. As illustrated in Figure 5, the uncovered range measures the portion of the x domain not covered by sampled layouts. A good selector should therefore minimize both the uncovered range and the largest local gap, $\max(\Delta x)$, while converging with few layouts.

To evaluate the runtime models, we use two complementary tests. First, we estimate in-distribution accuracy using 5-fold cross-validation on the dataset generated by each selector; we call this *self CV*. Second, we test the same model on the combined dataset produced by all selectors; we call this stricter setting *combined CV*. Unless otherwise stated, we report the maximal absolute prediction error across all benchmarks and CPU generations. More formally, for benchmark b running on machine m, let $e_{b,m}(x)$ denote the absolute runtime prediction error at x, and let $E^{max}_{b,m} = \max_x\{e_{b,m}(x)\}$ denote its maximum over all relevant x values. We quantify the success of each model and its underlying selector using $E_{max} = \max_{b,m}\{E^{max}_{b,m}\}$, the maximum error across all x values, benchmarks, and machines.

While it may be tempting to average the error across the different x values, doing so would be methodologically erroneous for the following reason. Consider a new virtual memory design D evaluated for benchmark b on machine m by a partial simulator S_D, which produces x_D. The runtime model will thus be queried at x_D only; no other x point is relevant to D. Notably, if x_D falls in the region where the model's prediction error is maximal, then, by definition, its absolute prediction error is $E^{max}_{b,m}$. Had we used the mean absolute prediction error across x values instead, it would have quantified the error associated with many other x values, unrelated to D, giving us the wrong result. The direct-segments example in Section 3 illustrates this case: its held-out proxy (the test sample) lies near the low x end of the domain where the maximal prediction error occurs.

Although averaging errors across different x values is methodologically erroneous, it *is* valid to average $E^{max}_{b,m}$ across benchmark/machine pairs. We thus also consider $E_{mean} = \text{mean}_{b,m}\{E^{max}_{b,m}\}$. Table 7 compares E_{max} to E_{mean} for the three-input, third-degree polynomial model with Ridge regularization (bars associated with "+regularization" in Figure 1).

metric	*BaselineS*	*PebsS*	*GeneticS*	*BayesianS*	*Moselect*
E_{max}	2,605%	5,227%	154,530%	16.6%	13.6%
E_{mean}	108%	184%	4,498%	5.0%	4.0%

Table 7. *E_{max} vs. E_{mean} across all selectors.*

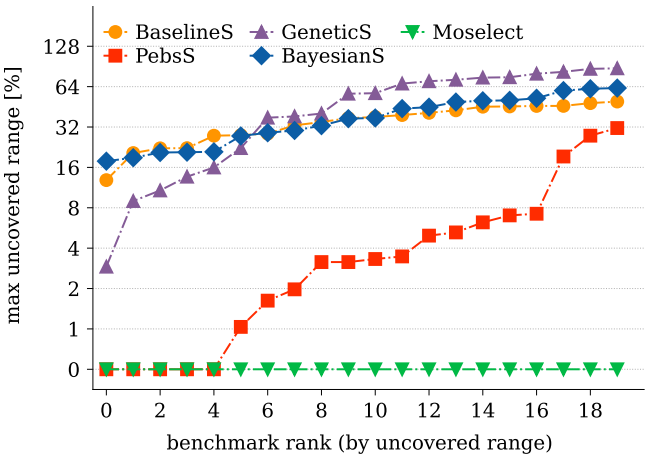


Figure 9. *Only Moselect achieves full coverage for all benchmarks.*

As can be seen, E_{mean} is lower than E_{max}, but the two metrics behave qualitatively similarly otherwise.

5.2 Coverage

Figure 9 compares the uncovered ranges of the x domain across all selectors. BaselineS typically leaves a substantial fraction of the domain uncovered because its heuristics do not adapt. PebsS is better but still suboptimal as it too is static and does not correct discrepancies between predicted and measured region effects. Despite being adaptive, BayesianS provides coverage similar to BaselineS, while GeneticS improves over BaselineS on some benchmarks but worsens on others. In contrast, Moselect achieves full coverage for all benchmarks, converging with 46 layouts on average; the other selectors exhaust their full budgets (up to 54) and do not to achieve this goal.

The BO results should not be interpreted as "failure." BayesianS uses a surrogate of the relationship between layouts and measured L2 TLB misses rather than explicitly closing gaps in the x domain. Its exploration-exploitation strategy may therefore leave broad walk-cycle intervals underexplored while still producing useful runtime-model samples.

5.3 Regularization Choice: Ridge Over Lasso

The stronger training sets generated by Moselect and BayesianS expose another limitation in the original modeling pipeline: regularization. BaselineM uses Lasso regression [25, 56] to fit the third-degree polynomial over (H, M, C). Lasso provides regularization and implicit feature selection, but its sparsity-inducing behavior can be problematic when these inputs are strongly correlated.

We observe strong correlations among (H, M, C), particularly between TLB misses and page-walk cycles. In such settings, Lasso may retain one correlated variable while suppressing others, reducing prediction accuracy [62]. We therefore refine Mosmodel by replacing Lasso with Ridge regression. Ridge shrinks correlated coefficients together rather than enforcing sparsity, making it more robust for these inputs [23, 59, 62]. As shown next, Ridge further reduces prediction error, although the larger gain comes from better selector-generated training data.

5.4 Prediction CV Error

Experimental Setup We now evaluate how selector choice affects runtime-model accuracy. Using Mosalloc, we execute each workload under the selected layouts, collect the counters required by Mosmodel, and build and evaluate the models using the methodology above.

Selector Quality Dominates Figure 10 reports the maximal absolute prediction errors across all workloads and platforms. Under self CV, several selectors appear reasonably accurate because they are tested on their own samples. Combined CV, however, exposes substantially larger errors for selectors other than Moselect, showing poor generalization to unseen layouts. In contrast, Moselect maintains much lower errors. Using the original Lasso-based BaselineM, replacing BaselineS with Moselect reduces the maximal error to 7.9% under self CV and 37.9% under combined CV. Since the regression method is unchanged, this improvement reflects the higher-quality training data generated by Moselect.

Why Coverage Matters The main reason for this accuracy gap is sample placement. Large uncovered regions provide little training support near those operating points, forcing the model to rely more on extrapolation. In contrast, denser and more uniformly distributed samples better support interpolation between nearby observations. Moselect therefore improves accuracy not by changing the model form, but by providing a better training set.

Coverage vs. Optimization-Driven Selection Bayesian optimization constitutes a different approach. Although the BayesianS selector leaves substantial gaps in the walk-cycle x domain (Figure 9), its layouts still produce relatively accurate runtime models. Its GP models $L \mapsto M(L)$ rather than explicitly targeting walk-cycle coverage, yet the selected layouts provide useful runtime-model samples. Thus, dense walk-cycle coverage is not the only path to accurate prediction. Moselect nevertheless achieves both stronger coverage and better prediction accuracy through domain-specific guidance and execution feedback.

Ridge Improves Robustness Replacing Lasso with Ridge improves robustness, particularly under combined CV (Figure 10). For Moselect, the maximal self CV error remains 7.9%, while the combined CV error decreases from 37.9% to

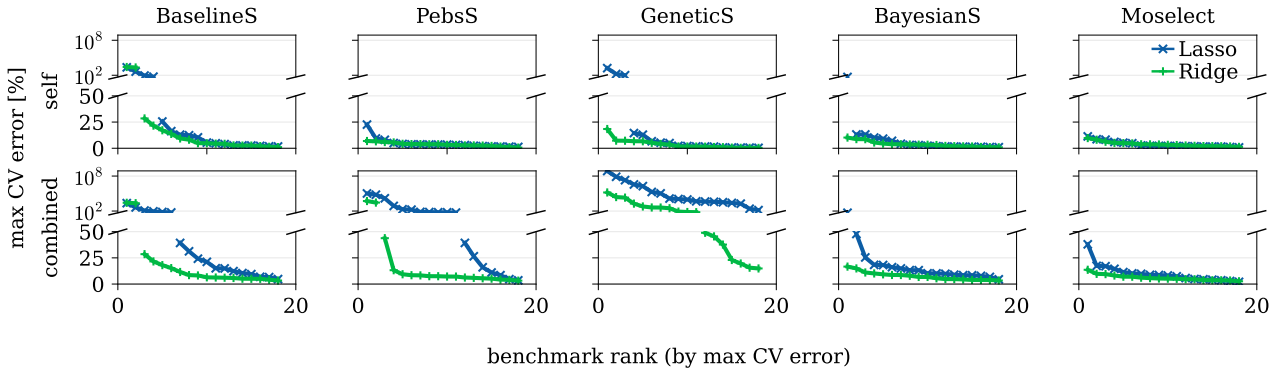


Figure 10. *Maximal absolute prediction error for the five selectors under self and combined CV, using Lasso and Ridge regression. Combined CV exposes substantially larger errors for selectors other than Moselect, while Ridge improves combined CV robustness.*

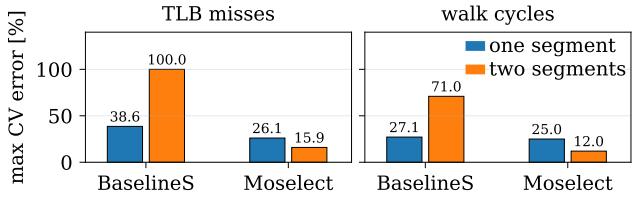


Figure 11. *Moselect supports more accurate piecewise linear models than BaselineS when using one or two segments.*

13.6%. This is consistent with Ridge's greater robustness to the correlated inputs (H, M, C). Nevertheless, regularization cannot compensate for the poorly placed samples produced by weaker selectors.

Takeaway Overall, better coverage is strongly associated with lower prediction error among the coverage-driven selectors. BayesianS is an informative exception: despite optimizing L2 TLB misses rather than walk-cycle coverage, its layouts still produce relatively accurate models. Moselect nevertheless achieves both the strongest x-domain coverage and the best prediction accuracy, with Ridge providing an additional model-side improvement.

5.5 Piecewise Linearity

We include errors for piecewise-linear models because Figure 4 may suggest that such a model could capture BaselineS samples well. However, Figure 11 shows that this does not generalize. Under *combined CV*, three-segment models perform markedly worse than one- and two-segment models, indicating overfitting to sparse local structure.

The contrast between BaselineS and Moselect highlights selector quality. For BaselineS, one segment outperforms two because large x-domain gaps leave the second segment supported by few samples. In contrast, Moselect achieves lower error with two segments because its denser coverage better supports the more expressive model. Thus, the benefit of piecewise linearity also depends on selector coverage.

5.6 Model Simplification

The higher-quality training data produced by Moselect also allow the runtime model and required simulator to be simplified without materially sacrificing accuracy. Starting from the refined Ridge-based Mosmodel, we reduce the polynomial degree from Poly3 to Poly2 and remove page-walk cycles C. The resulting Poly2 (H, M) model requires only TLB-level simulation rather than simulation of page walks and their cache-hierarchy interactions.

Figure 12 reports the complete sweep across polynomial degrees and input combinations. Models trained on Moselect data are substantially less sensitive to model configuration than those trained on BaselineS data. Several C-based configurations, including C and (M, C), also remain competitive when accuracy is prioritized over simulator complexity.

Figure 13 distills the main accuracy–complexity tradeoff for Moselect-generated data. Moving from Poly1 to Poly2 substantially reduces maximal prediction error, whereas Poly3 provides little additional benefit, particularly under combined CV. With Poly2, removing C increases the maximal combined CV error from 13.5% for (H, M, C) to 16.0% for (H, M), a 2.5-percentage-point increase, while reducing the required simulator from the TLB and cache hierarchy to the TLB alone. Thus, Poly2 (H, M) provides an attractive simplification point.

BaselineS behaves differently: its models are highly sensitive to the chosen inputs, and adding H can substantially increase error. For graph500/4GiB on Skylake (Figure 14), nearly all BaselineS layouts produce H values within roughly the lowest 10% of the observed range. Adding H therefore introduces a weakly covered dimension, increasing instability and overfitting. In contrast, Moselect provides sufficient coverage to support simpler and more stable models.

5.7 Self CV for Model Selection

Figure 13 also asks whether self CV can guide model simplification when the external data required for combined CV

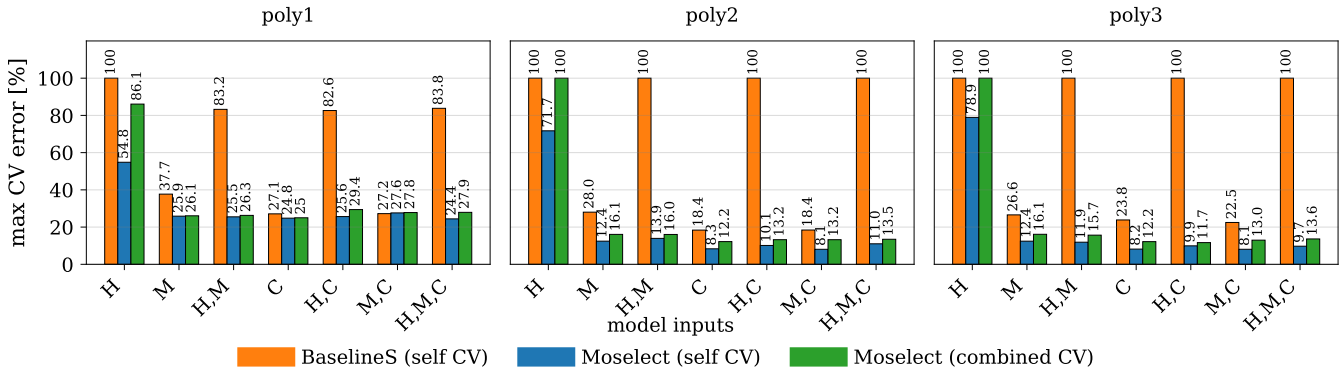


Figure 12. *Full model-simplification results across polynomial degrees and input sets. For BaselineS, we report self CV; for Moselect, we report self and combined CV. Moselect-generated data are substantially more stable across model configurations.*

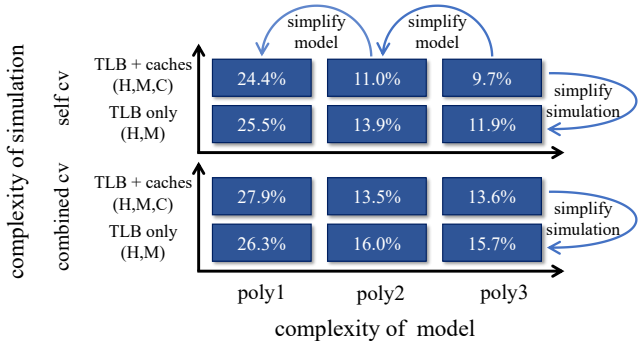


Figure 13. *Summary of tradeoffs between model complexity, simulator complexity, and prediction accuracy for Moselect-generated data.*

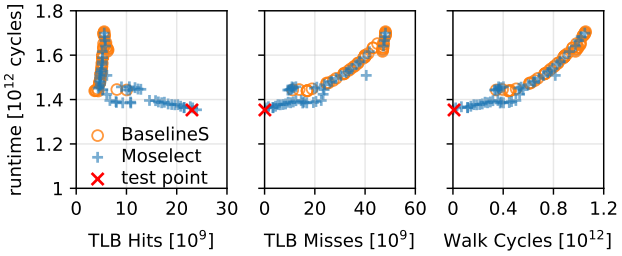


Figure 14. *BaselineS samples narrower counter regions—the most notable example is the L2-TLB hits (H), which hardly vary—resulting in less stable and accurate models. Moselect rectifies this problem.*

are unavailable. Researchers will typically have only their selector's dataset, so we compare self and combined CV along the same simplification path from (H, M, C) to (H, M).

Self CV exposes the same qualitative tradeoff as combined CV. For Poly2 (H, M), it captures the large gain over Poly1, while Poly3 provides little additional benefit. Combined CV reinforces this result, with Poly2 and Poly3 nearly indistinguishable for (H, M). Both therefore identify Poly2 (H, M) as an attractive accuracy–complexity point.

For Poly2 (H, M), the maximal error increases from 13.9% under self CV to 16.0% under combined CV. Thus, Moselect-generated data allow self CV to provide useful model-selection guidance, while combined CV remains the stricter test when cross-selector data are available.

6 Conclusions

Accurate model-based virtual memory evaluation depends on selecting effective layouts for runtime-model training. Moselect automates this selection, improving worst-case accuracy while enabling simpler models and simulators.

Acknowledgments

We thank our shepherd, Karthik Sriram, and the anonymous reviewers for their helpful feedback. We also thank Shaul Markovitch for suggesting the use of Gray-code encoding when implementing the Bayesian layout selector. This research was supported by the Israel Science Foundation (ISF), Grant No. 1137/22.

A Artifact Appendix

A.1 Abstract

The artifact provides the source code and experimental workflow for Moselect, our coverage-driven memory-layout selector for model-based evaluation of virtual-memory designs. It includes multi-window Mosalloc, native benchmark measurement, alternative selectors, and runtime-model analysis. A self-contained random-access toy benchmark operating on a 1 GiB array exercises the complete workflow without an external benchmark suite. The reduced test profiles the workload, constructs mixed 4 KiB/2 MiB layouts, collects hardware measurements, and generates a scatter plot. The published artifact release, `asplos27-v1.0`, is archived on Zenodo at https://doi.org/10.5281/zenodo.22685294. Paper-scale reproduction additionally requires the workloads and Intel platforms described in Section 5.1.

A.2 Artifact Summary

Algorithm: Moselect; PEBS, genetic, Bayesian optimization, and growing-, random-, and sliding-window selectors.

Program: Included toy benchmark; paper workloads are SPEC CPU2006, SPEC CPU2017, Graph500, XSBench, GUPS, and GAPBS.

Compilation: C/C++ compiler, GNU Make, CMake, and Python 3; binaries are built from source. No toy benchmark transformation is needed.

Model: Empirical polynomial runtime models fitted from native measurements, with Lasso and Ridge regularization.

Data set: None needed for the toy test; licensed SPEC software and inputs must be obtained separately.

Run-time environment: We used Ubuntu 20.04 on x86-64, Bash, kernel-compatible `perf`, Python packages in `requirements.txt`, and sudo privileges.

Hardware: Intel PMU/PEBS support and hugepage support; Broadwell or newer is recommended for the quick start. The paper uses Haswell, Broadwell, and Skylake.

Run-time state: Fixed frequency and NUMA placement; a lightly loaded or dedicated machine is recommended.

Execution: Repeated native runs under selected layouts; the basic test uses one repetition and at most 25 layouts.

Metrics: Runtime, L2 TLB hits/misses, page-walk cycles, coverage, maximum adjacent-sample gap, and model prediction error.

Output: Measurement files, selector state, model results, and PDF plots; basic-test output is `analysis/moselect/scatter.pdf`.

Experiments: Linux, GNU Make, Bash, Python, `perf`, Intel PEBS, and Mosalloc.

Disk space: Approximately 5 GiB for the basic workflow, excluding external benchmark suites.

Preparation/execution time: About 10 minutes to prepare and 15 minutes for the reduced test, depending on the machine. Full reproduction may require days per selector or weeks across all platforms.

Public availability: The archived release is openly available on Zenodo; source code is also available on GitHub.

Code/data licenses: Top-level code is GPL-3.0; third-party components retain their licenses. Generated data distributed with the artifact follow the repository terms; proprietary benchmark software is excluded.

Archived release: `asplos27-v1.0`; DOI 10.5281/zenodo.22685294.

A.3 Description and Dependencies

A.3.1 Access. For reproduction, download the release associated with https://doi.org/10.5281/zenodo.22685294, extract its ZIP archive, and enter the extracted artifact root. The archive includes Mosalloc; no separate submodule checkout is needed. The release README provides detailed setup, experiment, and troubleshooting instructions. The source-code repository is available at https://github.com/technion-csl/moselect. Use the archived release when reproducing or citing this work.

The `experiments/` directory contains selectors and experiment state; `analysis/` contains result processing; `mosalloc/` contains Mosalloc; `scripts/` contains measurement utilities; `predictors/` contains model utilities; and `toy_benchmark/` contains the supplied workload.

A.3.2 Hardware and Software. The processor must expose the PMU events used by the workflow and support Intel Precise Event-Based Sampling (PEBS). Event availability and encodings vary across processor generations and must be checked when using a different Intel platform. The toy benchmark needs a 1 GiB array plus memory for Linux, experiment processes, and a reserved 2 MiB hugepage pool; paper benchmarks need larger systems. The workflow uses `sudo` to reserve hugepages and configure system facilities. Linux `perf` must match the running kernel and be available on `PATH`. No pretrained model is needed: models are fitted from measured (H, M, C) values and runtimes.

A.4 Installation and Basic Test

From the artifact repository root, run:

```
make install-prereqs
source .venv/bin/activate
make test_artifact
```

The installation target provisions system dependencies and creates `.venv` with the packages in `requirements.txt`. The test uses `MOSELECT_NUM_OF_REPEATS=1`, `MOSELECT_NUM_LAYOUTS=25`, and `MOSELECT_MAX_GAP=8`. It exercises PEBS profiling, static initialization, adaptive selection, Mosalloc-based execution, PMU measurement, and result processing.

The test succeeds if it completes without an artifact error, collects valid measurements for multiple layouts, and produces a non-empty `analysis/moselect/scatter.pdf`. Compare this plot qualitatively with `toy_benchmark/broadwell_scatter.pdf` or `toy_benchmark/skylake_scatter.pdf`. Exact coordinates, slopes, axis ranges, and correlations can differ across machines. This basic test checks functionality; it does not reproduce the paper's numerical coverage or prediction-error results.

A.5 Experimental Workflow

The active workload is selected by `BENCHMARK_PATH` in `benchmark.mk`; the default is `toy_benchmark/`. Each workload directory supplies `pre_run.sh`, `run.sh`, and `post_run.sh` for preparation, measured execution, and cleanup. The workflow first determines the memory footprint and samples retired L2 TLB-miss addresses under the all-4 KiB layout. It aggregates samples into normalized 2 MiB region weights.

Static initialization partitions regions into PEBS-weight groups with targets of approximately 56%, 28%, and 14%. The eight group subsets and the all-2 MiB layout form nine anchors. Their measured walk cycles locate gaps in the normalized selector domain. Adaptive refinement adds or removes 2 MiB regions, executes each proposed layout, and uses its measured location to guide the next proposal. Selection stops at the maximum-gap target or layout budget. CSV-backed state, including `subgroups.log`, `layout_pages.log`, and per-subgroup logs, supports inspection of individual decisions.

A.6 Evaluation and Expected Results

A.6.1 Coverage and Convergence. For paper-scale evaluation, provide the desired benchmark wrapper and run:

```
make BENCHMARK_PATH=<benchmark-directory> \
     MOSELECT_NUM_OF_REPEATS=4 \
     MOSELECT_NUM_LAYOUTS=50 \
     MOSELECT_MAX_GAP=4 moselect
```

Inspect the maximum gap between adjacent measured walk-cycle samples and the generated coverage plots. On the evaluated workloads and platforms, Moselect reduces these gaps toward the configured 4% target. Selected layouts and convergence points need not be identical on a different machine. Quantitative reproduction requires the paper's benchmark inputs, processor configuration, and repetition count.

A.6.2 Runtime-Model Accuracy and Simplification. Use `cross_validate.py` with the collected measurement data. It uses scikit-learn's `RepeatedKFold` for repeated five-fold cross-validation and reports maximal absolute prediction error. For the simplification study, configure `features` to evaluate H, M, C, (H, M), (H, C), (M, C), and (H, M, C), comparing the polynomial degrees and Lasso/Ridge variants evaluated in the paper. On the paper's collected datasets, the original Lasso model trained on Moselect data yields maximum errors of 7.9% under self CV and 37.9% under combined CV; Ridge yields 7.9% and 13.6%, respectively. These are reference results for the paper's datasets, not acceptance thresholds for measurements collected on other systems. Interpret coverage and prediction-error differences in light of PMU support, processor generation, NUMA placement, frequency, hugepage configuration, kernel/toolchain versions, and background activity.

A.7 Customization and Limitations

Provide compatible benchmark wrapper scripts and set `BENCHMARK_PATH` to evaluate another workload; `client_server_example/` illustrates multi-process workloads. The layout budget, repetitions, and maximum-gap percentage can be changed independently using the variables above. Alternative selectors share the measurement infrastructure and analysis pipeline. The evaluated selector uses 4 KiB and 2 MiB pages; end-to-end 1 GiB-aware selection is outside this artifact's evaluated scope. SPEC CPU reproduction requires a separately licensed installation.

References

[1] Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. Mosalloc: Mosaic memory allocator, 2020. http://mosalloc.cs.technion.ac.il, http://github.com/technion-csl/mosalloc. (Accessed: April 2026).

[2] Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. Predicting execution times with partial simulations in virtual memory research: why and how. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 456–470. IEEE, 2020. http://doi.org/10.1109/MICRO50266.2020.00046.

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 476–487, 2012. http://doi.org/10.1145/2366231.2337214.

[4] Samira Mirbagher Ajorpaz, Elba Garza, Gilles Pokam, and Daniel A Jiménez. Chirp: Control-flow history reuse prediction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 131–145, 2020. http://doi.org/10.1109/MICRO50266.2020.00023.

[5] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019. http://doi.org/10.1109/ACCESS.2019.2917698.

[6] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2017. http://doi.org/10.1145/3079856.3080209.

[7] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 515–528, 2020. http://doi.org/10.1109/ISCA45697.2020.00050.

[8] AMD, Inc. *AMD64 Architecture Programmer's Manual, Volume 2*, 2013. http://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2. (Accessed: April 2026).

[9] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, Jeremiah Willcock, Anton Korzh, and Marcin Zalewski. Graph 500 benchmark specification. http://graph500.org/?page_id=12, 2017. Version 2.1 (Accessed: April 2026).

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 48–59, 2010. http://doi.org/10.1145/1815961.1815970.

[11] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013. http://doi.org/10.1145/2485922.2485943.

[12] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015. http://arxiv.org/abs/1508.03619.

[13] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. Scalable distributed last-level TLBs using low-latency interconnects. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 271–284, 2018. http://doi.org/10.1109/MICRO.2018.00030.

[14] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–73, 2011. http://doi.org/10.1109/HPCA.2011.5749717.

[15] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative TLB prefetchers for chip multiprocessors. In *ACM Architectural*

Support for Programming Languages and Operating Systems (ASPLOS), pages 359–370, 2010. http://doi.org/10.1145/1735971.1736060.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *CAN*, 39(2):1–7, August 2011. http://doi.org/10.1145/2024716.2024718.

[17] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 41–42, 2018. http://doi.org/10.1145/3185768.3185771.

[18] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A Jiménez, and Marc Casas. Instruction-aware cooperative TLB and cache replacement policies. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 619–636, 2025. http://doi.org/10.1145/3669940.3707247.

[19] Licheng Chen, Yanan Wang, Zehan Cui, Yongbing Huang, Yungang Bao, and Mingyu Chen. Scattered superpage: A case for bridging the gap between superpage and page coloring. In *IEEE International Conference on Computer Design (ICCD)*, pages 177–184, 2013. http://doi.org/10.1109/ICCD.2013.6657040.

[20] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 435–448, 2017. http://doi.org/10.1145/3037697.3037704.

[21] The Linux Kernel documentation. Transparent hugepage support. http://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2017. (Accessed: April 2026).

[22] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234, 2015.

[23] Solveig Engebretsen and Jon Bohlin. Statistical predictions with glmnet. *Clinical epigenetics*, 11(1):123, 2019.

[24] Peter I Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[25] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Springer Series in Statistics, 2001.

[26] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: A tool to instrument x86-64 TLB misses. *ACM SIGARCH Computer Architecture News (CAN)*, 42, 2014. http://doi.org/10.1145/2669594.2669599.

[27] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–189, 2014. http://doi.org/10.1109/MICRO.2014.37.

[28] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 707–718, 2016. http://doi.org/10.1109/ISCA.2016.67.

[29] Zhiqiang Gong, Ping Zhong, and Weidong Hu. Diversity in machine learning. *IEEE Access*, 7:64323–64350, 2019.

[30] Frank Gray. Pulse code communication. *United States Patent Number 2632058*, 1953.

[31] Faruk Guvenilir and Yale N. Patt. Tailored page sizes. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 900–912, 2020. http://doi.org/10.1109/ISCA45697.2020.00078.

[32] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News (CAN)*, 34(4):1–17, September 2006. http://doi.org/10.1145/1186736.1186737.

[33] John L. Henning. SPEC CPU2006 memory footprint. *ACM SIGARCH Computer Architecture News (CAN)*, 35(1):84–89, March 2007. http://doi.org/10.1145/1241601.1241618.

[34] John L. Henning. SPEC CPU2017 benchmark descriptions. http://www.spec.org/cpu2017/Docs/index.html, 2018. (Accessed: April 2026).

[35] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, 2015. http://tinyurl.com/intel-x86-3a. (Accessed: April 2026).

[36] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, 2015. http://tinyurl.com/intel-x86-3b. (Accessed: April 2026).

[37] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2016. http://tinyurl.com/intel-x86-optim. (Accessed: April 2026).

[38] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 66–78, 2015. http://doi.org/10.1145/2749469.2749471.

[39] David Koester and Bob Lucas. RandomAccess – GUPS (Giga updates per second). http://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html, 2004. (Accessed: April 2026).

[40] The Linux Kernel Wikis. *Perf Wiki*, June 2020. http://perf.wiki.kernel.org/index.php/Main_Page. (Accessed: April 2026).

[41] The Linux man-pages project. *mmap(2) man page*. http://man7.org/linux/man-pages/man2/mmap.2.html. (Accessed: April 2026).

[42] The Linux man-pages project. *perf - Performance analysis tools for Linux*, March 2021. http://man7.org/linux/man-pages/man1/perf.1.html. (Accessed: April 2026).

[43] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):2:1–2:38, April 2013. http://doi.org/10.1145/2445572.2445574.

[44] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. CSALT: Context switch aware large TLB. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 449–462, 2017. http://doi.org/10.1145/3123939.3124549.

[45] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1023–1036, 2019. http://doi.org/10.1145/3352460.3358294.

[46] Eric Munson. libhugetlbfs. http://github.com/libhugetlbfs/libhugetlbfs, 2015. libhugetlbfs README (Accessed: April 2026).

[47] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray User Group Conference (CUG)*, 19:45–74, 2010. http://www.richardmurphy.net/archive/cug-may2010.pdf.

[48] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. Prediction-based superpage-friendly TLB designs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 210–222, 2015.

[49] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using superpages to improve VIPT caches. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 193–206, 2018. http://doi.org/10.1109/ISCA.2018.00026.

[50] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 913–925, 2020. http://doi.org/10.1109/ISCA45697.2020.00079.

[51] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Efficient synonym filtering and scalable delayed translation for hybrid virtual caching. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 217–229, 2016.

[52] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. Increasing TLB reach by exploiting clustering in page translations. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, 2014. http://doi.org/10.1109/HPCA.2014.6835964.

[53] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2015. http://doi.org/10.1145/2830772.2830773.

[54] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 469–480, 2017. http://doi.org/10.1145/3079856.3080210.

[55] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. BabelFish: Fusing address translations for containers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 501–514, 2020. http://doi.org/10.1109/ISCA45697.2020.00049.

[56] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, 1996.

[57] Xin Tong and Andreas Moshovos. BarTLB: Barren page resistant TLB for managed runtime languages. In *IEEE International Conference on Computer Design (ICCD)*, pages 270–277, 2014. http://doi.org/10.1109/ICCD.2014.6974692.

[58] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *International Conference on the Physics of Reactors (PHYSOR)*, 2014. http://www.mcs.anl.gov/papers/P5064-0114.pdf.

[59] Diego Vidaurre, Concha Bielza, and Pedro Larranaga. A survey of L1 regression. *International Statistical Review*, 81(3):361–387, 2013.

[60] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *ACM SIGMETRICS International Conference on Measurement and Modeling (SIGMETRICS)*, pages 337–350, 2016. http://doi.org/10.1145/2896377.2901456.

[61] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct memory translation for virtualized clouds. In *ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 287–304, 2024. http://doi.org/10.1145/3620665.3640358.

[62] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 67(2):301–320, 2005.