Online Phase-Adaptive Data Layout Selection*

Chengliang Zhang[1] and Martin Hirzel[2]

[1] Microsoft in Redmond, WA; chengzh@microsoft.com
(C. Zhang was a student at the U. of Rochester when doing this work.)
[2] IBM in Hawthorne, NY; hirzel@us.ibm.com

Abstract. Good data layouts improve cache and TLB performance of object-oriented software, but unfortunately, selecting an optimal data layout a priori is NP-hard. This paper introduces layout auditing, a technique that selects the best among a set of layouts online (while the program is running). Layout auditing randomly applies different layouts over time and observes their performance. As it becomes confident about which layout performs best, it selects that layout with higher probability. But if a phase shift causes a different layout to perform better, layout auditing learns the new best layout. We implemented our technique in a product Java virtual machine, using copying generational garbage collection to produce different layouts, and tested it on 20 long-running benchmarks and 4 hardware platforms. Given any combination of benchmark and platform, layout auditing consistently performs close to the best layout for that combination, without requiring offline training.

1 Introduction

Cache and TLB misses often cause programs to run slowly. For example, we estimate that pseudojbb05 spends 34% of its time stalled in misses on a 4-processor AMD machine [17]. Cache and TLB misses often stem from a mismatch between data layout and data access order. For example, **Fig. 1** shows that the same layout can degrade or improve runtime depending on how well it matches the program's data accesses, and on how expensive the layout is to apply.

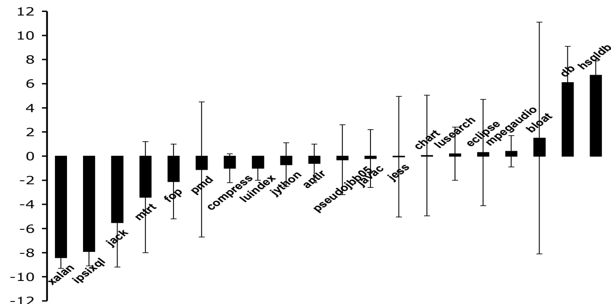


Fig. 1. Percent speedup of *HI* data layout, compared to *BF*, on an 8-processor AMD at 4× the minimum heap size. Section 5 explains *HI* and *BF*, and Section 6 explains the methodology. The error bars show 95% confidence intervals from Student's t-test.

* This research was funded in part by DARPA contract No. NBCH30390004.

Results like those in Fig. 1 are typical: optimizations that improve performance for some programs often risk degrading performance for other programs. The results depend on tradeoffs between optimization costs and rewards, and on interactions between complex software and hardware systems. Picking the best data layout a priori is difficult. Petrank and Rawitz showed that even with perfect knowledge of the data access order, finding the optimal data placement, or approximating it within a constant factor, is NP-hard [32]. Zhang et al. showed that finding a general affinity-hierarchy layout is also NP-hard [45]. While these hardness results were shown for fairly regular scientific code, reliance on pointers and dynamic dispatch exacerbate the problem further for object-oriented code. Practically, picking a data layout before the program starts would require training runs and command line arguments, both of which impede user acceptance.

This paper proposes *layout auditing*, a framework for picking the best data layout online without requiring any user input. Layout auditing optimizes data layouts with a *try-measure-decide* feedback loop: use a data reorganizer to try one of several data layouts, use a profiler to measure the impact of the data layout on performance, and use a controller to decide which layout to try next.

The data reorganizer *tries* a layout for the program's data. The data reorganizer can reorder data arrays or index arrays for scientific programs [11]; or it can copy objects in a specific order during garbage collection for object-oriented programs [13]; or it can even remap addresses using special-purpose hardware [46]. Layout auditing works with off-the-shelf data reorganizers [24], and the engineers who implement them need not be aware that the layouts get picked based on profile information.

The profiler *measures* the reward of the layout of the program's current data. The reward is high iff the program spends little physical time per virtual time. Virtual time is a data layout-independent measure of program progress, such as loop iterations, allocated bytes, or instructions. Physical time (seconds) depends on the data layout. The profiler can either simply obtain physical time from the CPU clock, or it can derive physical time from other information sources. The profiler reports not just the reward of a data layout in terms of program performance, but also the cost of the data reorganizer, profiler, and controller.

The controller *decides* the layout for the next data reorganization, and also decides how much, if any, time to spend on profiling. If the controller is confident about which layout is best, it picks that layout to exploit its good performance characteristics. If the controller is uncertain, it picks a layout it is curious about, to explore its reward. The controller uses off-the-shelf reinforcement learning techniques [41]. It turns the reward and curiosity for each data layout into a probability, and then picks randomly from its repertoire of layouts using those probabilities. To adapt to phase shifts, the profiler never allows probabilities to drop to zero, so that it always performs a minimal amount of exploration.

Selecting one of several layouts is a *multi-armed bandit* problem [33]. The analogy is that of a slot machine (one-armed bandit), but with more than one arm. Each arm is a data layout, and the reward is improved program performance. The controller repeatedly tries different arms, and hones in on the best

ones. Layout auditing subscribes to the philosophy of *blind justice*. The controller is a fair and impartial judge who decides based on hard evidence only, and gives each candidate the benefit of the doubt. In fact, the judge is not only fair, but also merciful: even when a layout performs badly, it still gets sampled occasionally to check for phase changes.

Layout auditing combines the advantages of two strands of prior work. First, like online profile-directed locality optimizations, it adapts to platforms, programs, and phases to achieve better performance than what offline optimization can achieve. Second, like performance auditing [26], it separates optimization concerns from controller concerns, it requires no correct model of complex hardware interaction, and it does not get fooled by misleading access patterns where finding the optimal data layout is NP-hard [32]. Unlike performance auditing, this paper addresses data layouts, not code optimization, and adapts to phases. This paper differs from prior profile-directed locality optimizations as well as from performance auditing in that it uses a uniform controller for not just performance rewards, but also optimization costs.

We evaluated layout auditing for 20 long-running Java programs on 4 hardware platforms. The layouts were produced by copying generational garbage collection changing the relative placement of heap objects in memory. Given any combination of benchmark and platform, layout auditing consistently performs close to the best layout for that combination.

Section 2 presents layout auditing as a framework, and Sections 3 to 5 present one concrete implementation that is evaluated in Sections 6 and 7. Section 8 sketches alternative implementations of the framework, Section 9 discusses related work, and Section 10 concludes.

2 Layout auditing framework

Fig. 2 illustrates the try-measure-decide feedback loop of layout auditing. The data reorganizer tries a data layout, the profiler measures its reward, and the controller decides the next actions of the data reorganizer and the profiler.

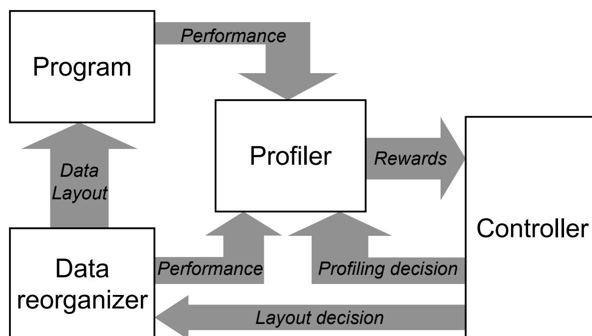


Fig. 2. Feedback loop.

2.1 Program

What: The program performs some calculation on behalf of the user. It is oblivious to the layout auditing feedback loop that surrounds it. The layout of the program's data in memory is determined by the data reorganizer, and the program's performance is monitored by the profiler.

How: Section 6 describes a suite of 20 large Java programs from a wide range of application domains. They run unperturbed on a product language runtime system with JIT compilation, a popular operating system, and stock hardware.

2.2 Data reorganizer

What: The data reorganizer executes a layout decision by placing the program's data in a specific order in memory. The layout affects program performance; in addition, the performance of the data reorganizer itself is monitored by the profiler.

How: Garbage collection is widely used to support robust object-oriented software. Section 5 uses copying garbage collection to implement the data reorganizer. This paper is based on high-performance implementations of two well-known collectors [7, 44] that ship with a product language runtime system, and some experimental collectors [17].

2.3 Profiler

What: The profiler monitors the performance of the program and the data reorganizer. It reports rewards for each data layout to the controller. Rewards measure physical time per virtual time. Virtual time is a metric of program progress that is independent of the data layout, such as loop iterations, allocated bytes, or executed instructions.

How: Section 4 describes the minimalist profiler. It simply looks at the machine clock to obtain physical time in seconds, and counts bytes allocated as virtual time. The minimalist profiler uses the most authoritative model of the interaction of data layouts with the hardware: concrete measurements of unperturbed execution.

2.4 Controller

What: The controller turns rewards of data layouts into decisions for which layout to try next, and how much profiling to do. The controller is responsible for optimizing overall performance, even when the program has phase changes.

How: Section 3 describes the softmax controller. It uses a simple reinforcement learning policy [41] to turn rewards into probabilities. The controller remembers historical rewards to avoid unstable decisions when there is noise, but it decays old rewards to adapt to phase changes.

3 Softmax controller

The controller turns data layout rewards from the profiler into layout decisions for the data reorganizer, and profiling decisions for the profiler. It does so by first turning rewards into probabilities, then deciding randomly based on those probabilities. The two main challenges for the controller are noise and phase changes: noise is random jitter in program behavior that the controller should ignore, and phase changes are systematic transitions in program behavior that the controller should adapt to. The controller in this section solves both challenges for a small fixed number of layouts while remaining reasonably simple.

3.1 Layout decision

In reinforcement learning, functions that map rewards to probabilities are known as *policies*. The softmax controller is named for the *softmax* policy [41]:

$$\Pr(\ell) = \frac{e^{reward(\ell)/\tau}}{\sum_{\ell'} e^{reward(\ell')/\tau}} \tag{1}$$

Equation 1 calculates $\Pr(\ell)$, the probability with which the controller will decide on layout ℓ for the next data reorganization. Layouts with higher rewards receive higher probabilities, since $e^{reward(\ell)/\tau}$ is larger. Before exponentiation, each reward is divided by a temperature τ. A high τ makes probabilities of different rewards more similar. A low τ emphasizes the reward differences in the probabilities; at low temperatures, controller decisions "freeze". The division in Equation 1 normalizes the probabilities such that they add up to 1.

Depending on the temperature, layout auditing will spend additional time exploring other layouts besides the best layout. Spending time on exploration is only justified if the information so far is too unreliable to exploit. To make this tradeoff, the controller computes the pooled standard error of the rewards of all layouts, and uses that as a *curiosity* value. The intuition for using error as curiosity is that when the error is high, the controller is curious to learn more, whereas additional data points will satisfy curiosity by shrinking the error. The controller sets the temperature such that the expected reward of the chosen layout differs from the reward of the best layout only by a small constant k times the curiosity. Given a temperature τ, the expected reward of a randomly chosen layout is

$$expectedReward(\tau) = \sum_{\ell} \left\{ \Pr_{\tau}(\ell) \cdot reward(\ell) \right\} \tag{2}$$

The controller tries different values for τ using binary search until the absolute difference between the maximum reward and the expected reward matches the desired target value $k \cdot curiosity$:

$$k \cdot curiosity = \left| \max_{\ell} \{reward(\ell)\} - expectedReward(\tau) \right| \tag{3}$$

We chose $k = 1\%$ to ensure performance close to the best layout.

Curiosity is the pooled standard error of historical rewards for different layouts. To adapt to changes in program behavior, it should weigh recent results more heavily than old results that might come from a different phase. The controller achieves this with exponential decay. In other words, the weight of a reward is $decay^{age}$, where the base *decay* can be for example 0.95, and the exponent *age* is the number of data reorganizations since the reward was measured. Because the statistical formula for pooled standard error does not directly accommodate weighing values, the controller implements exponential decay with a trick: it duplicates values with higher weights, then computes the statistics on a larger population.

To adapt to phase changes and to admit redemption after miscarriages of justice (if any), the controller shows mercy to layouts that seemed to perform badly in the past. It achieves this by assigning each layout a probability of at least 5%, regardless of its reward. The price of mercy is degraded performance compared with the best layout. The controller blindly assumes that all unexplored layouts, for which there is no data yet, initially have infinite rewards.

3.2 Profiling decision

Some profilers incur overhead, and should only be activated when their benefits (information gain) outweigh their costs (overhead). This paper uses a zero-overhead profiler, so the profiler is always active, without controller decisions. Nevertheless, this section offers a technique to the interested reader for controlling profiling overhead with layout auditing. The decision to profile ($p = \top$) or not ($p = \bot$) is a two-armed bandit problem, which the controller can decide with reinforcement learning analogously to the multi-armed layout decision.

The reward of profiling, $reward(p = \top)$, is the reward of satisfied curiosity, which Section 3.1 defined as the pooled standard error of layout costs. The reward of not profiling, $reward(p = \bot)$, is avoiding two overheads: profiling overhead incurred during program execution, plus overhead incurred when the profiler processes raw measurements to compute layout rewards.

The controller computes $reward(p = \top)$, and relies on the profiler to report its own overhead in the form of $reward(p = \bot)$. The controller then decides whether or not to profile during the next execution interval using the softmax policy

$$\Pr(p) = \frac{e^{reward(p)/\tau}}{\sum\limits_{p'} e^{reward(p')/\tau}} \qquad (4)$$

The temperature τ is the same as in Equation 3.

4 Minimalist profiler

The profiler monitors the performance of the program and the data reorganizer, and turns them into rewards for each data layout for the controller. The main

challenge for any profiler used in online optimization is maximizing truthfulness while ignoring noise and minimizing overhead and Heisenberg effects. If the controller should be a blind judge, then the profiler should be a reliable witness.

The measurements of the minimalist profiler are very simple: seconds and allocated bytes. Both can be obtained truthfully at negligible overhead. This section discusses how the minimalist profiler turns seconds and bytes into rewards for each layout. Internally, the minimalist profiler computes costs, which are negative rewards, so low costs correspond to high rewards and vice versa. A cost is a seconds-per-byte ratio, and has the advantage of being additive when there are different costs from different system components. Formally, the reward of a data layout ℓ is

$$\mathit{reward}(\ell) = -\mathit{cost}(\ell) \tag{5}$$

The cost of a layout ℓ is the sum of its execution time cost $\mathit{cost}_e(\ell)$ and its data reorganization cost $\mathit{cost}_r(\ell)$:

$$\mathit{cost}(\ell) = \mathit{cost}_e(\ell) + \mathit{cost}_r(\ell) \tag{6}$$

The quantities in Equation 6 represent averages of ratios of corresponding historical measurements. To explain what that means, we first introduce some notation. Let e_i be the physical time of the program execution interval that follows reorganization i; let v_i be the virtual time in number of bytes allocated between reorganizations i and $i+1$; and let ℓ_i be the layout of reorganization i. The minimalist profiler calculates

$$\mathit{cost}_e(\ell) = \mathrm{avg}\left\{\frac{e_i}{v_i} \mid \ell_i = \ell\right\} \tag{7}$$

In words: to compute the programs's execution time cost for layout ℓ, average the set of historical seconds per bytes ratios e_i/v_i that used layout ℓ. Likewise, given the physical time r_i of data reorganization number i, the formula for data reorganizer cost is

$$\mathit{cost}_r(\ell) = \mathrm{avg}\left\{\frac{r_i}{v_{i-1}} \mid \ell_i = \ell\right\} \tag{8}$$

The minimalist profiler assumes that reorganizer time r_i is proportional to the allocation volume v_{i-1} of the preceding execution interval, and that execution time e_i reflects the layout ℓ_i of the preceding data reorganization.

Averaging over historical values (Equations 7 and 8) reduces noise. To reduce noise further, the averages omit outliers. The averages are weighted toward recent data using an exponential decay curve, to adapt when program behavior changes over time.

In addition to rewards for layouts, profilers also report their own cost to the controller in the form of $\mathit{reward}(p = \bot)$, which is the reward of not profiling. Since the minimalist profiler incurs no overhead, there is no reward for not profiling, hence $\mathit{reward}(p = \bot)$ is always 0.

To summarize, the minimalist profiler uses only information that is trivially available on any platform: seconds and allocated bytes. The disadvantage is that layout auditing will settle slowly when there is too much noise. Another drawback is the assumption that program execution time reflects the data layout of the previous data reorganization only, which plays down the effect of data in a different memory area that was unaffected by that reorganization, and thus has a different layout. On the positive side, the minimalist profiler is cheap, portable, and direct.

5 Data reorganization with garbage collection

The data reorganizer tries a layout for the program's data. There are many possible implementations for data reorganizers; this paper chose to use off-the-shelf garbage collection algorithms [24]. This section reviews background on copying collectors, and describes some common data layouts.

5.1 Copying garbage collection

Copying garbage collection divides heap memory into two semispaces. Only one semispace is active for allocation at a time. Garbage collection starts when the active semispace is full. The collector traverses pointers from program variables to discover reachable objects, which it copies to the other semispace (from from-space to to-space). It updates all pointers to refer to the to-space copies, and discards the from-space originals. When the program resumes, it uses to-space as the active semispace for allocation. An example for a copying garbage collector is Fenichel and Yochelson's algorithm, which traverses objects with a recursive procedure [13].

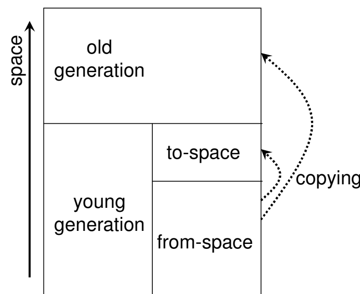


Fig. 3. Generational copying garbage collection.

Most language runtime systems today use generational garbage collectors, because they tend to yield the best throughput. Generational collectors segregate objects by age into generations [27, 42]. Younger generations are collected more often than older generations, which reduces overall collector work, because most objects become unreachable while they are still young. This paper is based on a

generational garbage collector with two generations, a copying young generation and a mark-sweep old generation. The collector also implements numerous other techniques, among others, parallelism [16] and tilted semi-spaces [28].

Fig. 3 shows the two kinds of copying: between the semi-spaces of the young generation, and from the young generation to the old generation. Each garbage collection can independently choose a copy order. Each set of objects allocated between the same two collections starts out in allocation order, and may then experience different layouts as it gets copied within the young generation. When the objects reach an age threshold, they get copied into the old generation (tenured), where they come to rest with a final layout.

The repeated data reorganizations when copying objects give layout auditing the opportunity to find the best layout.

5.2 Data layouts

This section briefly surveys some common data layouts; for a more comprehensive survey and evaluation, see [17].

Fenichel and Yochelson's recursive algorithm uses variables on the call stack to keep track of already copied objects that may contain pointers to not-yet copied objects [13]. Using a LIFO-stack leads to copying objects in depth-first (***DF***) order. Other *DF* copying collectors are not recursive, but maintain the stack as an explicit data structure and share it between parallel collector threads [8]. The *DF* layout is good for locality if the program often accesses a parent object together with a child object that it points to.

When the collector keeps objects in a FIFO-queue during the reachability traversal, it copies them in breadth-first (***BF***) order. Cheney's *BF* copying algorithm [7] uses the to-space copies of the objects themselves as an implicit queue. The *BF* layout is good for locality if the program often accesses sibling objects together.

An algorithm designed to achieve both the parent → child locality of the depth-first layout and the sibling locality of the breadth-first layout is hierarchical (***HI***) garbage collection by Moon and Wilson et al. [30, 44]. It works by copying a subtree to the same block as its root whenever possible.

Most of this paper uses layout auditing (***LA***) to choose between parallel implementations of the *BF* and *HI* layouts [22, 38].

Compacting collectors do not reserve an entire semispace for copying, instead they move objects toward one end of just one space. Sliding compaction aims at preserving the relative order of objects [24, Lisp 2 collector, Section 5.4]; [1, 14, 25]. When used as the sole copying mechanism, sliding compaction preserves allocation order (***AO***), which yields good locality when the program touches objects in the same order it allocated them, and can also facilitate stride prefetching [2, 23].

Segregating objects by *type* (***TY***) may yield better locality if the program tends to access objects of the same type together. This data layout also has potential benefits in reducing object header sizes, and has been used for reducing data reorganizer cost [19, 36].

6 Methodology

Name	Suite	Command line arguments	Description	PT	MB
antlr	DaCapo	`-s large -n 16`	parser generator	1	2.0
bloat	DaCapo	`-s large -n 4`	bytecode optimizer	1	16.1
chart	DaCapo	`-s large -n 8`	pdf graph plotter	1	14.3
compress	jvm98	`-a -m72 -M72 -s100`	Lempel-Ziv compressor	1	7.0
db	jvm98	`-a -m24 -M24 -s100`	in-memory database	1	11.2
eclipse	DaCapo	`-s small -n 4`	development environment	>1	14.0
fop	DaCapo	`-s large -n 60`	XSL-FO to pdf converter	1	9.1
hsqldb	DaCapo	`-s large -n 12`	in-memory JDBC database	20	173.8
ipsixql	Colorado	`80 7`	in-memory XML database	1	2.5
jack	jvm98	`-a -m164 -M164 -s100`	parser generator	1	1.3
javac	jvm98	`-a -m92 -M92 -s100`	Java compiler	1	20.5
jess	jvm98	`-a -m228 -M228 -s100`	expert shell system	1	2.1
jython	DaCapo	`-s large -n 4`	Python interpreter	1	1.9
luindex	DaCapo	`-s large -n 32`	text indexing for search	1	2.2
lusearch	DaCapo	`-s large -n 8`	keyword search in text	32	7.1
mpegaudio	jvm98	`-a -m156 -M156 -s100`	audio file decompressor	1	1.0
mtrt	jvm98	`-a -m232 -M232 -s100`	multi-threaded raytracer	2	8.7
pmd	DaCapo	`-s large -n 4`	source code analyzer	1	15.7
pseudojbb05	jbb05	`SPECjbb-4x200000.props`	business benchmark	4	123.9
xalan	DaCapo	`-s large -n 16`	XSLT processor	1	27.5

Table 1. Benchmark programs.

Table 1 shows the benchmark suite, consisting of 20 Java programs: pseudojbb05, which runs SPECjbb2005[3] for a fixed number of transactions; the 7 SPECjvm98 programs[4]; the 11 DaCapo benchmarks version 2006-10 [4]; and ipsixql[5]. Except for Fig. 4, all numbers in this paper are averages of nine JVM process invocations. Within each JVM invocation, the layout auditor starts with a clean slate, learning the best layout online as it goes. As is common practice for these benchmarks, each run contains several iterations (application invocations within one JVM process invocation), see Column "Command line arguments". Timings measure the entire run of the JVM process, and thus include any overheads incurred by layout auditing, such as initially making wrong decisions. Furthermore, all numbers in this paper are checked for statistical confidence using Student's t-test. Wherever the t-test indicates that performance differences are too small to be relevant at 95% confidence, we report a "0" value instead.

Column "PT" in Table 1 shows the number of program threads. The JVM also has some service threads that run concurrently with the program. Garbage collection is parallel with itself, but not concurrent with program execution. Column "MB" gives, in megabytes, the minimum heap size in which the program

[3] http://www.spec.org/jbb2005/
[4] http://www.spec.org/osg/jvm98/
[5] http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench/

runs without throwing an OutOfMemoryError. Most experiments in this paper provision each program with 4× its minimum heap size; garbage collection kicks in when the heap size is exhausted.

	L1 Cache		L2 Cache		TLB	
	AMD	Intel	AMD	Intel	AMD	Intel
Associativity	2	8	16	8	4	8
Block size	64 B	64 B	64 B	64 B	4 KB	4 KB
Capacity/blocks	1,024	256	16K	16K	512	64
Capacity/bytes	64K	16K	1,024K	1,024K	2,048K	256K

Table 2. Memory hierarchy parameters per core.

The experiments in this paper ran on one 2-processor Linux/IA32 machine, and on three different Linux/AMD machines with 2, 4, and 8 processors. We used the default run level of Linux (not single-user mode) to demonstrate that layout auditing can make correct decisions even in the presence of noise. The Intel machine was a Pentium 4 clocked at 3.2GHz with SMT, so the 2 physical processors correspond to 4 virtual processors. The AMD machines had Opteron 270 cores clocked at 2GHz, with 2 cores per chip. **Table 2** shows the data caches and TLBs for each core. We implemented layout auditing in J9, which is IBM's high-performance product Java virtual machine. The experiments in this paper are based on an internal development release of J9.

7 Results

This section evaluates data layout auditing using the concrete component instantiations from earlier sections: softmax policy, minimalist profiler, and data reorganization by copying garbage collection.

7.1 A control theoretic approach to controller evaluation

Layout auditing employs an online feedback loop to control a system. Such feedback loops have been extensively studied in control theory. Control theory commonly talks about *SASO* properties: Stability, Accuracy, Settling, and Overshoot. A good controller is a controller that is stable, accurately makes the right decisions, settles on that decision quickly, and does not overshoot the range of acceptable values. In the context of layout auditing, *stability* means sticking with a data layout once the controller picks one; *accuracy* means picking the data layout that yields the best performance; and *settling* is the time from the start or from a phase shift until the controller has made a decision. Overshoot does not apply in this context, because all layout decisions are in the range of acceptable values by definition. This is common for discrete, rather than continuous, control systems.

In addition to the SASO properties, layout auditing strives to achieve another desirable property: low overhead. Since the minimalist profiler treats the time for

data reorganization as part of the reward of a data layout, there is no separate overhead for data reorganization. The minimalist profiler just reads the clock and counts bytes, so it does not incur any overhead on its own. This leaves controller *overhead*: time spent doing the statistical calculations in the softmax controller. On average, each control decision takes on the order of 0.1ms. Compared to data reorganization times, which are on the order of 10ms to 100ms, controller overhead is negligible in most cases.

Phase adaptivity is the ability of the controller to change its decision if the program changes its behavior such that a different data layout becomes the best data layout. Phase adaptivity is another way to look at settling, accuracy, and stability. The minimalist profiler and the softmax controller achieve phase adaptivity by using exponential decay to forget old profile information. The decay factor determines how well layout auditing can adapt to phase changes.

Overall, layout auditing can make investments, such as profiling overhead, data reorganization cost, or time spent exploring data layouts it is curious about. For these investments, it reaps rewards, such as improved program execution time or improved data reorganization time due to reduced cache and TLB misses. The success of layout auditing depends on its ability to make the right tradeoff between the different investments and rewards.

7.2 Accuracy

This section explores the accuracy of the layout auditor presented in this paper. Accuracy is the ability of the controller to accurately pick the correct data layout. If it does, then the bottom-line performance of a program when run with layout auditing should match the performance of that program with its best statically chosen layout. In terms of Fig. 1, layout auditing should get all the speedups for programs at the right side of the bar chart, while avoiding all the slowdowns for programs at the left side of the bar chart. To evaluate accuracy, this section ran all 20 benchmark programs from Table 1 using the breadth-first (*BF*) and hierarchical (*HI*) data layout, both with and without layout auditing (*LA*).

Table 3 shows the results. For each of the 4 runtime platforms (2-processor Intel and 2-, 4-, and 8-processor AMD), there is one column for each of the data layouts *BF* and *HI* and one for layout auditing *LA*. All the numbers are percent slowdowns compared to the best runtime of the given benchmark/platform combination. For example, for ipsixql on the 2-processor Intel machine, *BF* was best, *HI* caused a 12% slowdown compared to *BF*, and *LA* matched the performance of *BF*. A “0” in Table 3 means that the results of the 9 runs with that data layout were indistinguishable from the results of the 9 runs with the best data layout for that benchmark and platform, using Student’s t-test at 95% confidence. The bottom of Table 3 shows summary rows: “Average” is the arithmetic mean of the slowdowns of the layout compared to the best layout for each benchmark, “# not 0” counts benchmarks for which the layout was not the best, and “Worst” is the maximum slowdown of the layout compared to the best.

Table 3 demonstrates that on all four platforms, online layout selection performs almost as well as an oracle that would pick the best layout for each program

Benchmark	Intel-2			AMD-2			AMD-4			AMD-8		
	BF	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*
antlr	0	1.1	2.1	1.4	0	2.2	0	0	0	0	0	0
bloat	0	0	0	0	0	0	0	0	0	0	0	0
chart	0	0	0	0	0	0	0	0	0	0	0	0
compress	0	1.2	0	0	0	0	0	0	0	0	0	0
db	5.2	0	2.9	6.0	0	1.9	0	0	0	6.5	0	0
eclipse	0	0	0	0	0	0	0	0	0	0	0	0
fop	0	0	0	0	0	0	0	0	0	0	0	0
hsqldb	0	0	0	0	0	0	2.8	0	0	7.2	0	0
ipsixql	0	12.0	0	0	10.7	1.9	0	10.4	0	0	7.9	1.4
jack	0	0	0	0	2.4	0	0	0	0	0	5.5	0
javac	0	1.5	0	0	1.3	0	0	0	0	0	0	0
jess	0	1.4	2.2	0	3.6	3.1	0	0	0	0	0	0
jython	0	0	0	0	0	0	0	0	0	0	0	0
luindex	0	0	0	0	0	0	0	0	0	0	1.0	1.0
lusearch	0	0	0	0	0	0	0	2.7	0	0	0	0
mpegaudio	0	1.8	0	0	0	0	0	0	0	0	0	0
mtrt	0	0	0	0	0	0	0	0	0	0	0	0
pmd	0	0	0	0	0	0	8.4	0	0	0	0	0
pseudojbb05	2.1	0	1.2	1.6	0	0	0	0	0	0	0	0
xalan	0	0	0	0	0	0	0	4.0	0	0	8.4	4.3
Average	0.4	0.9	0.4	0.5	0.9	0.5	0.6	0.9	0.0	0.7	1.1	0.3
# not 0	2	6	4	3	4	4	2	3	0	2	4	3
Worst	5.2	12.0	2.9	6.0	10.7	3.1	8.4	10.4	0.0	7.2	8.4	4.3

Table 3. Percent slowdown compared to best, on varying platforms at heap size 4×.

offline. Note that Petrank and Rawitz have shown conclusively that building such an offline oracle would be impractical [32]. Layout auditing usually, but not always, matches the performance of the best data layout for a program and platform; sometimes the program finishes executing too quickly for *LA* to settle on the best layout and recoup its exploration costs. However, layout auditing has the most benign worst cases. Statically picking the wrong layout can slow down execution by up to 12.0%, but dynamically picking with layout auditing never causes slowdowns exceeding 4.3%.

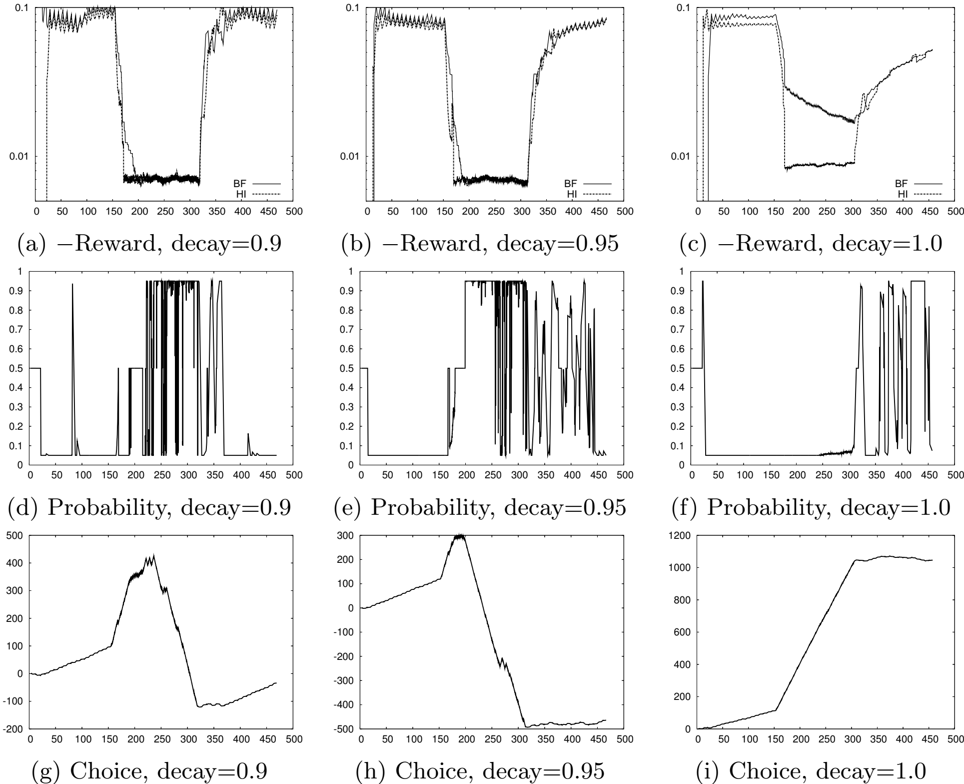
As noted in prior work [20, 38], some benchmarks, such as db and ipsixql, are unusually sensitive to data layouts. For those programs, layout auditing has the largest benefits. But it is equally important that for benchmarks that are mostly insensitive to data layouts, layout auditing does not degrade performance appreciably. Except for antlr and jess, this is usually the case. The reliable accuracy of layout auditing over a large range of programs and platforms gives it an edge over traditional locality optimizations.

To summarize, layout auditing is accurate. It makes good on its promise of requiring no model of the complex hardware/software interaction: it works equally well with no user tuning on four platforms.

7.3 Settling, stability, and phase adaptivity

This section investigates how long our implementation of layout auditing takes to settle, whether it is stable once it reaches a decision, and whether it can adapt to phase changes. This section answers these questions with a layout auditing experiment designed to illustrate phase changes, while still being realistic. Let T be the time in seconds since the start of the run, then the experiment first executes benchmark db from $T = 0$ to $T = 155$, then executes benchmark mtrt from $T = 155$ to $T = 320$, and finally goes back to db from $T = 320$ to $T = 475$. The softmax controller decides between the breadth-first data layout *BF* and the hierarchical data layout *HI*.

Benchmark db is much more data layout sensitive than mtrt. This constitutes a challenging scenario for settling, stability, and phase adaptivity. The two data layouts *BF* and *HI* have been shown to exhibit among the largest performance differences between common data layouts [17, Figure 4]. The experiment ran on the 2-processor AMD machine, and used heap size 44.8MB, which is 4× the minimum for db and 5.1× the minimum for mtrt. This setup models what happens when a server machine changes to a different workload that exercises different code.



(a) −Reward, decay=0.9 (b) −Reward, decay=0.95 (c) −Reward, decay=1.0

(d) Probability, decay=0.9 (e) Probability, decay=0.95 (f) Probability, decay=1.0

(g) Choice, decay=0.9 (h) Choice, decay=0.95 (i) Choice, decay=1.0

Fig. 4. Controller behavior over time for db/mtrt/db run, on AMD-2.

Fig. 4 shows the results. There are three columns: Column (a/d/g) is based on a run where the minimalist profiler and the softmax controller use decay 0.9, Column (b/e/h) uses decay 0.95, and Column (c/f/i) did not decay historical values (decay=1.0). The x-axis of all graphs is the same: physical time in seconds. Row (a/b/c) shows rewards as reported by the minimalist profiler, Row (d/e/f) shows the controller's current probability of *BF*, and Row (g/h/i) shows the cumulative number of decisions for *HI* and against *BF*. Each time the controller chooses *HI* for a data reorganization, the choice curve increases by one; each time the controller chooses *BF*, the choice curve decreases by one.

Rewards are physical time per virtual time, where lower is better. The reward graphs (Figures 4(a/b/c)) use a logarithmic y-axis, because data layout rewards are an order of magnitude higher in db than in mtrt. The phase transitions at around $T = 155$ and $T = 320$ are clearly visible. With a decay value of 0.9, the minimalist profiler quickly forgets earlier data, and therefore computes a reward that closely follows each spike in the data. Zooming in closely on the first phase in Fig. 4(a) reveals that the rewards for *HI* are higher than the rewards for *BF*, but the difference is lower than the amplitude of the program's own performance behavior over time. Fig. 4(c) shows that when the decay is 1.0, the profiler never forgets, and the curve becomes smooth over time. This means that without decay, the profiler can not adapt to phases: by the last phase, the rewards of *BF* and *HI* are indistinguishable. Figures 4(a/b/c) show that the controller faces a tough challenge: it has to learn the best layout despite the fact that the difference between the layouts is dwarfed by the difference between program phases.

The probability graphs (Figures 4(d/e/f)) illustrate *settling*. For decay 0.9, the controller settles on the best data layout for each phase at approximately $T = 20$, $T = 220$, and $T = 370$, which is 20, 75, and 50 seconds after the phase transitions. For decay 0.95, the controller settles on the best data layout for each phase at approximately $T = 15$, $T = 200$, and $T = 445$, which is 15, 45, and 125 seconds after the phase transitions. For decay 1.0, the controller settles on *HI* for the first phase, but then takes the entire second phase to discover that *HI* is no longer the best, and is then unstable during the last phase. This illustrates that decay is necessary for *phase adaptivity*.

The probability graphs (Figures 4(d/e/f)) also illustrate *stability*. Fig. 4(d) shows that for decay 0.9, the controller is mostly stable during the first and the third phase, but has some spikes indicating instability. During the second phase, it is less stable, but Fig. 4(g) shows that it still chooses the correct layout most of the time. Fig. 4(e) shows that for decay 0.95, the controller is more stable during the first and second phases than with decay 0.9, but takes so long to settle that it only becomes stable again at the very end of the third phase. Fig. 4(f) shows that decay 1.0 leads to the best stability for stationary programs, at the cost of sacrificing settling after phase changes.

The choice graphs (Figures 4(g/h/i)) follow the probability graphs (Figures 4(d/e/f)), in the sense that when the probability is 50/50, the choice forms a horizontal line; when the probability for *HI* is high, the line rises; and when

the probability for *BF* is high, the line falls. Figure 4(i) may look unexpected at first. During the second phase, *BF* is the best layout, yet the choice curve rises even more steeply than during the first phase where *HI* was the best layout. The reason why it rises is that the controller makes wrong decisions: without decay, it fails to adapt to the phase change. The explanation why the curve rises more steeply is that there are more data reorganizations per second. That is caused by the fact that mtrt has a higher alloc/live ratio than db (see Table 4 in [4]). That also explains the increased gradients in Figure 4(g/h).

To summarize, this section showed settling times ranging from 15s to 125s for decay values of 0.9 and 0.95. Lower decay values lead to less stable controller decisions; when the decay is too small, the controller gets confused by noise in the program behavior. But in the other extreme, when the decay is too high or when there is no decay, the controller can not adapt to phase changes. This motivates why we designed the controller the way we did. With decay, the softmax controller adapts to phase changes by accurately picking the best data layout for each phase.

7.4 Cache and TLB behavior

This section explores whether the performance improvements of layout auditing come from reducing the program's data cache and TLB miss rates. The cache and TLB misses are measured in the same experiments as those for Section 7.2, by using PAPI [5] and then accumulating the counts for all program threads but excluding the data reorganizer.

Table 4 shows the results. The columns and rows are the same as in Table 3, and the numbers show percent miss rate increases compared to the layout with the best miss rate for a given program and platform. It turns out that layout auditing does **not** always achieve the lowest miss rate. We did not expect that it would: we already saw that layout auditing achieves the best performance, but program cache and TLB miss rates are only one factor in that. They have to be weighed against other performance factors, such as data reorganization time, hardware prefetching effects, instruction level parallelism, bus bandwidth, etc. Layout auditing does prevent the worst-case miss rates that occur for some programs; without layout auditing, those miss rate increases can easily amount to 100% and more. But more importantly, no matter how complex the performance effects of a particular hardware, layout auditing consistently and reliably optimizes the bottom-line performance.

7.5 Data reorganization cost and heap sizes

Switching between data layouts changes not just program performance, but also data reorganizer performance. In addition, if the data reorganizer is a garbage collector, its performance is also affected by the heap size: in a large heap, the program can allocate more memory before triggering a garbage collection. This usually means that objects have more time to die, and thus garbage collection

Benchmark	L2:Intel-2			L2:AMD-2			L2:AMD-4			L2:AMD-8		
	BF	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*
antlr	0	0	0	5.2	0	1.9	2.5	0	3.0	2.6	0	2.5
bloat	0	0	0	12.9	0	5.8	13.2	0	0	12.7	0	5.4
chart	0	0	0	0	0	0	0	0	0	0	0	0
compress	0	0	0	0	0	0	0	0	0	0	0	0
db	0	20.9	10.3	0	0	0	0	0	0	0	0	0
eclipse	0	0	0	0	0	0	0	0	0	0	0	0
fop	0	0	0	3.7	0	0	3.7	0	0	11.2	0	0
hsqldb	0	0	0	0	0	0	0	2.5	0	0	0	0
ipsixql	0	11.7	0	0	46.5	0	0	61.3	3.3	0	62.3	7.0
jack	0	0	0	0	0	13.5	0	0	0	0	14.2	0
javac	0	0	0	7.3	0	6.3	5.8	0	0	6.0	0	0
jess	0	0	0	0	0	0	0	7.2	5.2	0	9.1	3.7
jython	0	0	0	0	0	0	0	0	0	0	0	0
luindex	0	0	0	4.2	0	2.5	0	2.1	0	0	1.9	2.0
lusearch	0	0	0	0	0	0	0	12.5	0	0	0	0
mpegaudio	0	0	0	0	0	29.2	0	0	0	0	0	0
mtrt	7.0	0	0	0	0	7.9	7.1	0	0	0	0	0
pmd	0	0	0	0	0	0	0	0	0	10.2	0	0
pseudojbb05	0	0	0	9.2	0	4.8	8.7	0	5.5	8.2	0	5.0
xalan	0	0	0	4.1	0	2.3	3.4	0	0	0	0	0
Average	0.4	1.7	0.5	2.5	2.4	3.9	2.3	4.5	0.9	2.5	5.0	1.5
# not 0	1	2	1	7	1	9	7	5	4	6	5	7
Worst	7.0	20.9	10.3	12.9	46.5	29.2	13.2	61.3	5.5	12.7	62.3	7.0

Benchmark	TLB:Intel-2			TLB:AMD-2			TLB:AMD-4			TLB:AMD-8		
	BF	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*
antlr	0	0	0	2.0	0	0	0	0	0	0	0	0
bloat	10.1	0	0	0	0	0	5.7	0	0	0	0	0
chart	0	0	0	0	0	0	0	0	0	18.1	0	9.1
compress	0	0	0	0	0	1.1	0	0	0	0	0	0
db	134.2	0	59.7	163.3	0	46.2	167.3	0	65.3	177.6	0	94.3
eclipse	0	0	0	0	0	0	0	0	0	0	0	0
fop	0	0	0	1.7	0	2.0	4.0	3.9	0	0	0	0
hsqldb	13.5	0	0	12.6	0	0	11.0	0	0	0	0	0
ipsixql	20.5	22.9	0	31.2	0	0	0	0	0	0	11.4	0
jack	0	0	0	0	0	0	0	0	0	0	0	0
javac	15.9	0	13.3	14.9	0	9.4	11.0	0	12.1	9.0	0	0
jess	3.2	0	0	3.1	0	0	0	0	0	0	0	0
jython	10.4	0	0	0	0	0	0	0	0	0	0	0
luindex	9.3	0	0	0	0	0	0	0	0	0	0	0
lusearch	0	0	0	0	0	0	2.6	0	0	9.2	0	0
mpegaudio	0	0	0	0	0	0	0	2.9	2.1	0	0	0
mtrt	31.9	0	14.8	25.0	0	16.6	22.8	0	9.6	15.4	0	0
pmd	0	0	0	10.2	0	9.7	0	0	0	22.9	0	0
pseudojbb05	6.1	0	0	11.7	0	7.5	13.4	0	7.7	11.5	0	7.4
xalan	12.0	0	6.7	7.1	0	6.3	5.7	0	0	9.5	0	0
Average	14.1	1.2	5.0	14.9	0.0	5.2	12.8	0.4	5.1	13.7	0.6	5.5
# not 0	11	1	4	11	0	8	9	2	5	8	1	3
Worst	134.2	22.9	59.7	163.3	0.0	46.2	167.3	3.9	65.3	177.6	11.4	94.3

Table 4. Percent miss rate increase compared to best, at heap size 4×.

is cheaper, since it has to run less frequently and processes relatively fewer survivors.

Layout auditing takes data reorganization cost into account, as described in Section 4. It should therefore always find the right performance tradeoff irrespective of heap size and garbage collection cost. The experiments in this section provision each program with 2×, 4×, or 10× the minimum heap size in which the program runs without throwing an OutOfMemoryError. The tight heap size 2× (50% utilization) frequently triggers garbage collection; the standard heap

Benchmark	2× heap			4× heap			10× heap		
	BF	*HI*	*LA*	*BF*	*HI*	*LA*	*BF*	*HI*	*LA*
antlr	0	1.0	2.9	1.4	0	2.2	0	0	2.1
bloat	0	0	0	0	0	0	0	0	0
chart	0	0	0	0	0	0	0	0	0
compress	0	0	0.7	0	0	0	0	0	0
db	5.3	0	2.6	6.0	0	1.9	6.4	0	2.5
eclipse	0	0	0	0	0	0	0	0	0
fop	0	0	0	0	0	0	0	0	0
hsqldb	0	0	0	0	0	0	0	0	0
ipsixql	0	10.6	3.9	0	10.7	1.9	0	13.8	6.5
jack	0	2.3	0	0	2.4	0	0	0	0
javac	0	3.3	0	0	1.3	0	0	0	0
jess	0	2.1	4.1	0	3.6	3.1	0	0	0
jython	0	0	0	0	0	0	0	0	0
luindex	0	0	0	0	0	0	0	0	0
lusearch	0	0	0	0	0	0	0	0	0
mpegaudio	0	0	0	0	0	0	0	0	0
mtrt	0	0.7	0	0	0	0	0	0	0
pmd	0	0	0	0	0	0	0	0	0
pseudojbb05	0	0	0	1.6	0	0	0	0	0
xalan	0	1.0	0	0	0	0	0	0.8	0
Average	0.3	1.1	0.7	0.5	0.9	0.5	0.3	0.7	0.6
# not 0	1	7	5	3	4	4	1	2	3
Worst	5.3	10.6	4.1	6.0	10.7	3.1	6.4	13.8	6.5

Table 5. Percent slowdown compared to best, on AMD-2 at varying heap sizes.

size 4× (25% utilization) is what most of the rest of this paper uses; and the roomy heap size 10× (10% utilization) infrequently triggers garbage collection. **Table 5** shows the relative overall performance of the different data layouts in different heap sizes. Table 5 is organized similarly to Table 3, with which it shares the AMD-2 / 4× heap columns. Remember that "0" values indicate that whatever performance degregation there may be compared to the best layout is too small to be deemed statistically relevant. The 10× heap makes benchmarks less layout sensitive, since that heap size deemphasizes data reorganizer cost, one of the factors in performance differences. Conversely, the 2× heap makes benchmarks more layout sensitive. The slightly worse results in the 10× heap might be caused by fewer garbage collections offering fewer trials. Even so, in all three heap sizes, layout auditing picks the best layout most of the time.

7.6 Bandits with more than two arms

The results so far are based on layout auditing chosing between two layouts *BF* and *HI*. These are the only two high-performance layouts implemented in our infrastructure. So in order to explore chosing between three layouts, we had to resort to slow experimental garbage collectors. We use the algorithms from [17], but whereas that paper focuses on program time excluding data reorganization cost only, here we look at the total cost including data reorganization. We use the

Benchmark	AMD-2				AMD-8			
Benchmark	BF_s	DF_s	TY_s	LA_s	BF_s	DF_s	TY_s	LA_s
antlr	4.8	0	24.3	4.4	3.3	0	20.1	2.7
bloat	0	0	18.3	0	0	0	26.4	0
chart	0	0	10.3	0	0	0	79.9	0
compress	0	0	0	0	0	0	1.2	0
db	8.1	0	22.7	3.4	10.1	0	21.8	7.0
eclipse	0	0	18.2	7.2	0	0	24.5	8.8
fop	0	0	10.4	0	0	0	18.5	0
hsqldb	10.2	0	214.0	24.5	17.0	0	253.1	22.5
ipsixql	19.8	0	144.3	7.4	32.9	0	158.6	4.3
jack	0	2.4	14.7	0	0	0	13.4	0
javac	8.4	0	90.3	3.9	11.5	0	92.5	0
jess	0	0	17.6	2.4	0	0	16.4	4.6
jython	0	0	0	0	2.2	0	1.9	0
luindex	0	0	9.1	0	0	0	8.4	1.7
lusearch	0	0	5.9	0	0	0	13.5	7.0
mpegaudio	0	1.5	0	0	1.8	0	0	0
mtrt	0	0	151.6	0	0	0	212.4	0
pmd	0	0	35.3	0	0	0	39.2	5.5
pseudojbb05	2.6	0	74.9	9.3	9.4	0	113.6	18.9
xalan	5.5	0	15.8	6.9	8.5	0	47.6	4.9
Average	3.0	0.2	43.9	3.5	4.8	0.0	58.2	4.4
# not 0	7	2	17	9	9	0	19	11
Worst	19.8	2.4	214.0	24.5	32.9	0	253.1	22.5

Table 6. Multi-layout percent slowdown compared to best, at heap size 10×.

layouts BF_s, DF_s, and TY_s, where the subscript denotes slow implementations. Note in particular that $BF_s \neq BF$.

Table 6, which is formatted similarly to Table 3, shows the results. Despite running in a loose heap, the slow data reorganizer of TY_s causes high overheads compared to the other two reorganizers. Layout auditing successfully avoids the risk of degrading performance nearly as much as TY_s, and comes close to BF_s and DF_s. In fact, for most programs, layout auditing performs close to the best of the three layouts. We do not expect layout auditing to perform well with tens or hundreds of layouts, because the settling time would grow unreasonably long.

8 Alternative layout auditing components

This section discusses alternative data reorganizers, profilers, and controllers that fit in the layout auditing framework from Section 2.

8.1 Alternative data reorganizers

Layout auditing is designed to accommodate a variety of off-the-shelf data reorganization techniques. Section 5 already mentioned several data layouts (depth-first, breadth-first, hierarchical, allocation order). Other garbage collectors segregate objects by size, type, or allocating thread. One could even consider a

random data layout; while random layouts are unlikely to perform best, they are equally unlikely to perform worst, and can thus effectively prevent pathological interference situations.

While layout auditing works with profile-oblivious data layouts, it can be applied just as easily to decide whether or not to use profile-directed approaches, such as Huang et al.'s online object reordering [20] or the locality optimizations by Chen et al. [6].

As mentioned earlier in this paper, layout auditing is not confined to garbage collection; a variety of other data reorganizers has been proposed. One technique is to reorder data arrays or index arrays for scientific programs [11]. Zhang et al. present and simulate a piece of hardware that can remap data to a different layout [46]. Another possibility is to change the data layout during allocation, for example, by using different alignments, or by profile-directed techniques [37].

8.2 Alternative profilers

In the easiest case, the profiler just measures seconds by looking at the clock. The advantage is that this causes no overhead, but the disadvantage is that it makes it hard to isolate the impact of the data layout from the impact of extraneous effects. To complicate things further, it is often desirable to isolate the impact of the layout of some memory subspace from the impact of the layout of other subspaces. This challenge could be addressed with a *subspace locality profiler*.

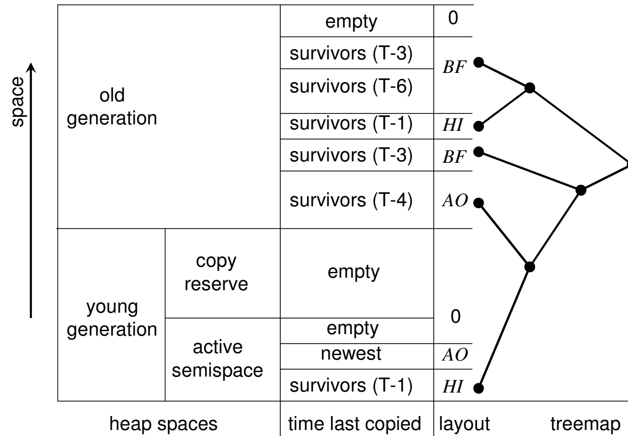


Fig. 5. Mapping from addresses to layouts.

For example, if the data reorganizer is a generational garbage collector (like in Section 5), each collection of the young generation copies some objects within the young generation, and others from the young to the old generation. Over time, a situation like in **Fig. 5** arises. The left column shows the heap spaces: an old generation, and a young generation with two semispaces. The middle

column further divides spaces into groups of objects, annotated by the last time they were copied; e.g., "survivors (T-3)" were tenured 3 collections ago, whereas the "newest" objects were allocated after the last collection and have yet to be copied for the first time. Column "layout" shows which copy order, and hence which data layout, the corresponding objects have. It is easy to keep track of the mapping from addresses to memory areas and their data layouts; a subspace locality profiler could do so with a treemap. What is needed, then, is a measurement of locality for specific data addresses.

One possibility for this is PEBS (precise event based sampling), where hardware counter overflows generate interrupts, and the interrupt handler can inspect parts of the machine state. Adl-Tabatabai et al. used PEBS on Itanium to identify objects that cause cache misses [2]. Similarly, one could count misses separately for memory subspaces with different data layouts. Unfortunately, on IA32 and AMD, the PEBS machine state does not contain the data address, and each interrupt costs several thousand cycles.

Another possibility for a subspace locality profiler is trace-driven cache simulation. To accommodate layout auditing, the tracing and simulation must occur online and automatically. Bursty tracing [3, 18] can produce a memory access trace at low overhead. Online cache simulation has been reported by Zhao et al. [48]. To use online trace-driven simulation for layout auditing, map simulated accesses and misses to data layouts via the treemap from Fig. 5.

A drawback of more sophisticated profilers is that they make more assumptions about how the software and hardware interact. Such assumptions can be misleading: for example, more cache misses do not necessarily imply worse performance if instruction-level parallelism overlays them with useful computation.

8.3 Alternative controllers

Layout auditing is designed to accommodate a variety of off-the-shelf machine learning techniques. The authors come from a systems background, and have to refer the reader to the relevant literature for details [41]. This paper uses a softmax policy. Other possibilities include sequential analysis and reinforcement computation.

Also, there are alternatives for dealing with phase changes. This paper uses exponential decay of historical profile information. Another possibility is to remember a sliding window of values. There are also more sophisticated standalone phase detectors [31, 35].

9 Related work

Layout auditing uses an online feedback loop that first tries different data layouts, then evaluates their performance, and based on that, changes data layout decisions later in the same run. This section reviews other online try-measure-decide feedback loops.

Lau et al. proposed performance auditing [26], which first tries different ways to optimize a method using a JIT compiler, then evaluates their performance,

and finally decides which one to use. Performance auditing addresses measurement noise by continuing to collect information until it reaches statistical confidence for a decision. Our work is also performance auditing, but we apply it to data, not code, and we extend it to adapt when program behavior changes over time.

We picked copying garbage collection as the mechanism for executing data layout decisions. Our controller switches between different copying collectors for the young generation. Soman et al. showed how to switch between a more diverse set of collectors, including both generational and non-generational, copying and non-copying algorithms [40]. But whereas we decide to switch to another collector based on online measurements of application performance, Soman et al. decide based on heap size thresholds.

Chen et al. try a data layout optimization in a garbage collector, measure whether it reduces miss rates, and throttle it if it does not [6]. Whereas Chen et al. use online feedback to throttle an optimization in one collector, we use online feedback to pick between multiple alternative collectors. Chen et al.'s throttling mechanism is woven into the collector, and this tightly integrated design compromises desirable features for both: the collector is not parallel, and the controller does not use statistical or machine learning techniques to deal with noise or with changes in program behavior.

Besides changing the data layout, an alternative technique for improving locality is prefetching. Some papers propose online try-measure-decide feedback loops for picking the best prefetch distance [34, 47].

Zhang et al. use an online try-measure-decide feedback loop for picking the largest memory footprint that does not yet cause paging [45]. Unlike our work, they change the heap size, not the data layout, and focus on paging, not on cache and TLB locality.

A number of papers show how to pick between differently optimized versions of scientific code at runtime [12, 15, 43]. Unlike our work, and unlike Lau et al.'s performance auditing [26], they pay little attention to dealing with noise. Also, they focus on code, we focus on data.

Other online data layout optimizations do not use a try-measure-decide feedback loop [9, 10, 20, 21]. Instead, they use online profile data to predict which layout will benefit performance, without checking later whether the prediction came true. Petrank and Rawitz showed that these predictions can be easily fooled by misleading data access patterns [32]. This is exacerbated by the fact that the predictions rely on a model of hardware/software interactions, which change and become more complex over time.

Reinforcement learning has been used for programming language optimizations in the past (e.g., [29]), and different machine learning techniques have been used for selecting garbage collectors [39]. But unlike layout auditing, these approaches require offline training runs, and their benefits only become available by providing learned information to a second production run.

10 Conclusions

Layout auditing is an approach for picking the best among a set of data layouts by trying them and measuring their performance. It handles noise with a continuous feedback loop, and with a controller based on reinforcement learning. It smoothly adapts when program behavior changes over time. It controls its own profiling and exploration overheads by tuning them in the same way it tunes data layout decisions. This paper demonstrates that layout auditing achieves close to the best performance for all programs we tried it on, and avoids pathological worst cases that can happen with any statically chosen layout. Layout auditing successfully reevaluates its decisions after phase changes during program execution.

The trend towards multicore machines is likely to increase the importance of locality optimizations in the near future, as more CPUs compete for limited memory subsystem resources. At the same time, hardware complexity is on the rise, and the unpredictability of hardware behavior calls for approaches like layout auditing that optimize regardless of the detailed instruction-level behavior.

Acknowledgements: We thank Matthew Arnold, Jeremy Lau, Rodric Rabbah, Erik Altman, Priya Nagpurkar, and the anonymous reviewers for their feedback.

References

1. D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.
2. A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Programming Language Design and Implementation (PLDI)*, 2004.
3. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Programming Language Design and Implementation (PLDI)*, 2001.
4. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.
5. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *IEEE SuperComputing (SC)*, 2000.
6. W. K. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Programming Language Design and Implementation (PLDI)*, 2006.
7. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM (CACM)*, 1970.
8. P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Programming Language Design and Implementation (PLDI)*, 2001.
9. T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

10. R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM (CACM)*, 1988.
11. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Programming Language Design and Implementation (PLDI)*, 1999.
12. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Programming Language Design and Implementation (PLDI)*, 1997.
13. R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM (CACM)*, 1969.
14. C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium (JVM)*, 2001.
15. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *High Performance Embedded Architectures & Compilers (HiPEAC)*. LNCS 3793, 2005.
16. R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 1985.
17. M. Hirzel. Data layouts for object-oriented programs. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.
18. M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001.
19. M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.
20. X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.
21. A. Ibrahim and W. R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *European Conference for Object-Oriented Programming (ECOOP)*, 2006.
22. A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 1993.
23. T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *Programming Language Design and Implementation (PLDI)*, 2003.
24. R. Jones and R. Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.
25. H. Kermany and E. Petrank. The Compressor: Concurrent, incremental, and parallel compaction. In *Programming Language Design and Implementation (PLDI)*, 2006.
26. J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Programming Language Design and Implementation (PLDI)*, 2006.
27. H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 1983.
28. P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *International Symposium on Memory Management (ISMM)*, 2006.

29. A. McGovern, J. E. B. Moss, and A. G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning*, 49(2-3), 2002.
30. D. A. Moon. Garbage collection in a large Lisp system. In *LISP and Functional Programming (LFP)*, 1984.
31. P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V. Rajan. Online phase detection algorithms. In *Code Generation and Optimization (CGO)*, 2006.
32. E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Principles of Programming Languages (POPL)*, 2002.
33. H. E. Robbins. Some aspects of sequential design of experiments. *Bulletin of the American Mathematical Society*, (58):527–535, 1952.
34. R. H. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Parallel Architectures and Compilation Techniques (PACT)*, 1996.
35. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques (PACT)*, 2001.
36. Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Principles of Programming Languages (POPL)*, 2002.
37. Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.
38. D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *International Symposium on Memory Management (ISMM)*, 2006.
39. J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *International Symposium on Memory Management (ISMM)*, 2007.
40. S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *International Symposium on Memory Management (ISMM)*, 2004.
41. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.
42. D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 1984.
43. M. J. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Principles and Practice of Parallel Programming (PPoPP)*, 2001.
44. P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in a garbage-collected system. In *Conference on Programming Language Design and Implementation (PLDI)*, 1991.
45. C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Principles of Programming Languages (POPL)*, 2006.
46. L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 2001.
47. W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Code Generation and Optimization (CGO)*, 2006.
48. Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *Code Generation and Optimization (CGO)*, 2007.