Instance-Optimized Mapping with Portfolio Methods

Yibo Zhao, Panagiotis Manolios, and Cheng Tan
Northeastern University

Abstract

Mappings are ubiquitous in computer systems, such as translating virtual memory to physical memory, file paths to inode numbers, database keys to data locations. Traditional system mappings are often hand-crafted and data-agnostic. In this paper, we explore the use of neural networks as *learned mappings* that are automatically generated and data-dependent, optimizing performance for specific workloads and scenarios. Unlike prior learned structures, we employ a portfolio method consisting of a set of independent neural networks, each responsible for making sole decisions. Our preliminary results indicate that these portfolio mappings can generalize across multiple applications.

ACM Reference Format:
Yibo Zhao, Panagiotis Manolios, and Cheng Tan , *Northeastern University* . 2024. Instance-Optimized Mapping with Portfolio Methods. In *3rd Workshop on Practical Adoption Challenges of ML for Systems (PACMI '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3694715.3699982

1 Introduction

Mappings associate a set of data with another set of data. In computer systems, they are fundamental building blocks, and they are everywhere. For example, virtual memory maps virtual addresses to physical addresses; directory mappings translate paths to inodes (a data structure representing files and folders); database indexes get database keys and return data positions; packet classification maps network packets to the actions the routers need to take.

All of these are examples of mappings in systems. Though having the same functionality in principle (namely mapping), these mappings' implementations are vastly different. Virtual memory uses radix trees (i.e., 4/5-level page tables) and is implemented in hardware. The directory mapping uses hierarchical trees (i.e., the file system namespace); Databases indexes use many data structures, like B-Tree, hash table, and bitmap. Packet classification uses Trie and sometimes others.

It makes a lot of sense to tailor mappings for different systems because these mappings require different performance, have different hardware constraints, and some need to support advanced operations (e.g., prefix query, range query). We call these mappings tailored for different systems, *system-optimized mappings*.

In this paper, we want to go one step further: we want to optimize mappings for each *instance*—a user, an application, a workload, or even a period of time (e.g., during the peak load of a day). We call these mappings *instance-optimized mappings*. Why do people want instance-optimized mappings? The answer is performance. Instances have their own characteristics and exploiting these characteristics usually offer unprecedented performance. Another way to see this is that a system-optimized mapping exploits the properties of a particular system, so it works better for this system than others; whereas an instance-optimized mapping further exploits the characteristics of a particular instance, so it works even better for the given instance (e.g., a workload). People have observed this in different setups before [5, 9, 11, 12, 15].

However, implementing instance-optimized mappings in a traditional way—like how we build system-optimized mappings before—does not work, for two reasons. First, it will be too expensive and not scalable. People cannot afford the engineering efforts to tailor algorithms and data structures for every single instance. Second, humans are unable to capture the sophisticated heuristics that an instances carries.

Therefore, we need an approach that (a) automatically builds the mappings without too much human efforts, and (b) discovers complex instance-specific heuristics for better performance. In this paper, we propose such an approach which constructs a learned mapping based on neural networks, called *portfolio mappings*. Portfolio mappings adapt to different systems and learn heuristics from data with minimum human efforts.

The basic idea is as follows. A set of neural networks learns the instance's mapping independently. Each network covers a fraction of the input space. When serving a query, a selector picks a network that can handle the query; the picked network produces an output; and a tuner calibrates the network's output. Figure 1 overviews an abstract mapping, a classic directory mapping, and a portfolio mapping.

Portfolio mappings differ from existing learned structures [3, 4, 13, 19] in a combination of three design choices:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PACMI '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3699982

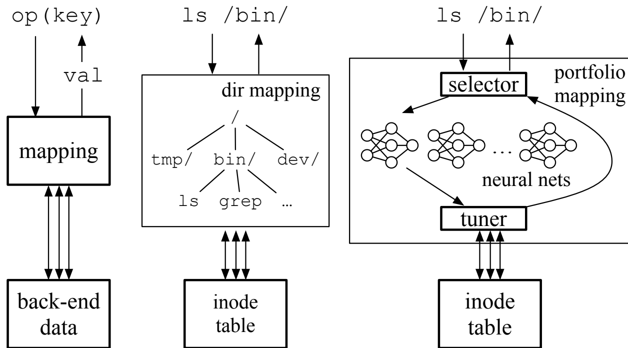

Figure 1. Mapping overview. An abstract mapping is depicted on the left, a traditional directory mapping in the middle, and a portfolio mapping on the right.

① compared with prior work, portfolio mappings use non-trivial neural networks[1]; ② neural networks are independent; and ③ every network makes sole decisions.

These lead to unique properties that portfolio mappings offer. First, compared with existing learned structures, portfolio mappings are versatile and can operate on mappings from multi-dimensional inputs to multi-dimensional outputs ($\mathbb{R}^n \rightarrow \mathbb{R}^n$) without requiring prior assumptions about the underlying systems. This is due to our choice ①, using non-trivial neural networks. Second, seemingly, portfolio mappings are similar to ensemble models [6]: both comprise a set of models. In fact, they are fundamentally different. Ensemble models rely on "crowd wisdom" to make decisions, whereas portfolio mappings' networks make sole decisions (choice ③). Finally, portfolio mappings are update-friendly. Since all neural networks are independent (②) and each serves a fraction of the input space, people can partially update the mapping when facing data drifting. All these benefits of course come with a cost: the training time of portfolio mappings is significantly higher than prior work, which sometimes could take three hours (§4).

2 Setup, definition, and goals

Setup. In the use cases of mappings in computer systems, there are three roles: an application, a mapping, and a piece of back-end data. An application (like a file system) sends a query *op* and its input x (e.g., open a file) to a mapping. The mapping executes $op(x)$ and returns to the application an output y (e.g., an inode number). In this process, the mapping will interact with the back-end data (e.g., the inode table on disk) multiple times to locate y.

Take the classic Unix file system as an example. When a program calls `open(/tmp/a/b,...)`, the file system walks the directory mapping: the file system starts from the root "`/`", loads the inode of "`/`" from the inode table, then looks for "`tmp`" in the content of the folder "`/`", and gets the inode number of the folder "`tmp`". Next, the file system repeats this process of searching the inode of "`a`" and "`b`". Finally, the file system returns the inode number of the file "`b`". Instead of walking a tree, portfolio mappings learn the path-to-inode mapping directly via neural networks (§4).

[1] As we will show later, the models we use are fully-connected feed-forward neural networks, which, while more sophisticated than linear models and other simple models used by existing learned structures, are still considered "simple" from a machine learning perspective.

In this paper, we define the mapping used in computer systems as follows.

Definition 1 (Mapping in systems)**.** A mapping is a function f, such that

$$f : \mathcal{Q} \times \mathcal{X} \rightarrow \mathcal{Y}$$

where: $\mathcal{Q}$ is a set of *query* types; $\mathcal{X}$ is the *domain*: for example, a set of strings, integers, floats, or arrays of previous primitive types; and $\mathcal{Y}$ is the *range*: for example, a set of integers, booleans or floats.

Consider the above directory mapping. A dir mapping in a file system has $\mathcal{X}$ = strings (i.e., paths) and $\mathcal{Y}$ = integers (i.e., inode numbers pointing to the inode table).

Portfolio mappings support three types of queries (*op*):

1. *prefix-query*: query all items that share the same prefix, for example, listing all files under a folder in a file system.
2. *range-query*: query all items within a given range, for example, getting students whose ages are between 10 and 12 in a database.
3. *point-query*: query the item indexed by a key, for example, querying the rule that should apply to a packet in a network swtich, or returning the value of a key in a key-value store.

Goals. A portfolio mapping is optimized for an instance. For the targeted instance, we expect that (a) the portfolio mapping has good performance: it has higher throughputs or lower latencies compared with classic data-agnostic mappings, and (b) the portfolio mapping is smaller in size: it should consume fewer memory than traditional data structures.

Compared with existing learned structures [3, 4, 13, 16], we expect portfolio mappings to be more general: they support different input types (e.g., multi-dimensional inputs) and various operations (e.g., prefix-query) in computer systems.

Non-goals. We do not expect portfolio mappings to replace traditional data structures. For one, there are limited scenarios when applications are bottlenecked by mappings. For another, the performance squeezed by portfolio mappings is at the cost of training time. If a mapping is used infrequently or is constantly changing, the benefits may not cover the efforts.

We do not expect portfolio mappings to outperform specialized learned structures in domains that they are designed for. Take database index as an example. Many existing learned indexes [4, 9] are highly optimized for the 1D-input-to-1D-output mapping problem via linear models. Portfolio mappings use neural networks which are fundamentally less efficient than linear models in this problem. Neural networks however are more general.

3 Portfolio mapping preview

Below is a preview of portfolio mapping: we describe how a portfolio mapping works using an example. We also briefly introduce how a portfolio mapping is trained.

Overview. Portfolio mapping has three main components: a *selector*, a *tuner*, and a set of neural networks.

The selector encodes an input (like a path) into a tensor, chooses a neural network, and feeds the tensor to the network. The input-tensor encoding is designed by developers for each of the systems. As an example, for the directory mapping, we split a path by "/" into an array of tokens, each of which is a string representing a file or a folder. Then, we apply a hash function to translate each token to a floating-point number $\in (0, 1)$. The picked network then does an inference for the input tensor and produces an output tensor (a one-dimensional tensor in the file system example). Finally, given the network's output, the tuner will binary search on the back-end data (e.g., an inode table) for the true result of the query.

Executing a dir-mapping query: an example. To illustrate how portfolio mapping works, we use a toy example of directory mapping to execute `open("/tmp/abc/def", ...)`. Figure 2 depicts the workflow.

To locate the file `/tmp/abc`, the selector first encodes the path into a tensor `[0.373,0.68,0,...,0]`, where "`tmp`" and "`abc`" are hashed into `0.373` and `0.68`, respectively. The length of the tensor is fixed for each file system (in our case, 20-dimensional tensors); any outliers with >20 nested folders will be handled using a fallback traditional approach.

Next, the selector picks a neural network based on its knowledge of which network satisfies which input domain. In our example, it chooses the left-most network in Figure 2. The network then outputs `[0.797]` which is a normalized position pointing to the slot `14346` on the inode table.

Finally, the tuner fetches the disk block containing `14346` from the disk, and conducts binary search to locate the targeted inode named "`/tmp/abc`" (the inode table is sorted). If the target inode is close to `14346`, like the case in Figure 2, then the loaded block (now in memory) likely contains the wanted inode, hence portfolio mapping does not have to further interact with the disk. Otherwise, portfolio mapping needs to load more blocks from the disk.

Training and composing. Building a portfolio mapping has two phases: a training phase and a composing phase.

The training phase creates a set of neural networks that learn the data distribution of a system's mapping. The expectation is that each network covers some subsets of the input space and all the trained networks together cover the entire space. We use brute-force testing to understand whether a given network supports an input segment; namely, if the network's outputs are within an error bound. The training finally produces a set of candidate networks with their supported segments.

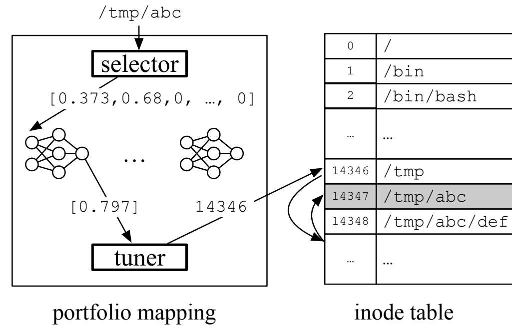

Figure 2. The workflow of portfolio mapping serving a query. The mapping will return inode `14347` (the shaded entry).

The raw candidate networks contain much redundancy. To save memory, a composing procedure selects a small subset of candidate networks that still covers the entire input space. Selecting the minimum subset is an optimization problem. We use a heuristic algorithm to construct this subset. Given the selected networks, we construct a selector for each portfolio mapping. The selector is a responsible table: an array that splits input space into segments, and each segment corresponds to one network which covers this input segment.

4 Preliminary results

We answer the following questions:

- How long does it take to inference the portfolio mapping structure compare to the baselines?
- How long does it take to build portfolio mapping structure compare to the baselines?

Setup. All experiments are conducted using a modified version of the SOSD benchmark [14], a standard benchmark suite for learned index. We extend SOSD to accommodate multidimensional key datasets for file system and packet classification applications. For model inference, we run experiments on a desktop with an Intel i7-12700K CPU, 16 GB memory, and Arch Linux with Kernel 6.9.7. For training, we use a PowerEdge R750XA server equipped with Intel Xeon Gold 6342 CPUs, 1TB memory, NVIDIA H100 GPUs, and Ubuntu 22.04. We use one GPU to train models in the following experiments.

Applications and benchmarks. We apply portfolio mappings to three applications: integer database index, filesystem path resolution, and network packet classification.

- *Integer database index:* Integer database indices map integer keys to integer positions within the database, facilitating the retrieval of values. All keys are assumed to be unique and sorted in ascending order. We use `Books` and `Wiki` databases from SOSD [14] with 10 million keys.
- *Filesystem path resolution:* For the file path resolution, paths (strings) map to inode numbers (integers); each path points to a unique inode number. We collect 100K file paths from an Arch Linux machine and covert the collected paths into 20-dimensional tensors.

Mapping	App	Dataset	Runtime (inference)	Size (KB)	Build time (s)
Portfolio mappings	DB index	Books	110ns (94ns)	383	1303
		Wiki	170ns (132ns)	3940	11408
	File path	Arch-fs	152ns (114ns)	2497	1486
	Packet	ClassBench	185ns (146ns)	3415	3037
RMI	DB index	Books	139ns (–)	3.08	0.34
		Wiki	262ns (–)	3.08	0.33
BTtree	DB index	Books	195ns (–)	139321	0.04
		Wiki	196ns (–)	139321	0.04
SIndex	File path	Arch-fs	189ns (–)	6.08	25.84

Figure 3. Runtime, mapping size, and build time for various mapping structures. The column "Runtime (inference)" shows the end-to-end query time, with the time spent on model inference for portfolio mappings indicated in parentheses.

- *Packet classification:* Network components, such as routers, often need to classify packets based on their five-tuple headers—source IP, destination IP, source port number, destination port number, and protocol. A packet classification structure maps each five-tuple header to a corresponding rule, identified by a rule ID (an integer). We use ClassBench [17], a standard benchmark for packet classification, to generate 5K rules.

Baselines. To demonstrate that portfolio mappings outperform traditional indices, we compare them with BTrees. We use the STX B+ tree implementation from SOSD [14]. In addition, we also compare portfolio mappings with existing specialized learned structures: RMI [10], designed for integer indices, and SIndex [19], designed for string (e.g., path) indices.

Preliminary results. For database index, we use two integer key datasets from the SOSD benchmark: `Books`, which contains book sales popularity data from Amazon [1], and `Wiki`, which includes Wikipedia article edit times [2]. Each dataset is composed of 10 million unsigned 4-byte integer keys, down-sampled from the original datasets included in SOSD. For each dataset, we conducted 1 million random key lookups with a batch size of 1 using a single thread.

For file system path evaluation, we collected real file paths by traversing the file system of a Arch Linux 6.9.7 machine. We use the `MD5` hash function to convert file names and directory names into 4-byte integers. Each file path is then encoded as a 20-dimensional integer vector. We handle longer paths separately, as they represent only a small number of cases. Each file path maps to a unique inode number. During experiments, we query the portfolio mappings for the inode number of each path at an random order.

For packet classification, we employ ClassBench [17] to generate 5K packet filtering rules. We then generate 100K randomly sampled IP addresses based on these rules for inference. For this experiment, we assume the rules do not overlap, with each sampled IP address corresponding to a unique rule.

We use a single GPU to train portfolio mappings. The mapping architecture is consistent across all three applications, comprising fully-connected feed-forward neural networks with two hidden layers with 32 neurons per layer. Figure 3 shows the preliminary results for all mappings.

In our experiments across three applications, portfolio mappings achieve performance comparable to specialized learned structures and surpass traditional indexes like B-Trees. Additionally, portfolio mappings generalize to all three applications without the need for specialized customization.

5 Discussion, future work, and conclusion

Pros and cons of learned mappings. Compared to traditional data structures, learned mappings—including portfolio mappings—have an unfair advantage: they spend time studying the data through training and possess prior knowledge when serving queries. Traditional mapping data structures, on the other hand, are data agnostic.

Being data-aware fundamentally has pros and cons. The benefit is that the learned data structures work exceptionally well with the data distribution they have been trained on. However, the drawback is that if the data changes, updating these structures can be challenging and expensive. We design portfolio mappings to balance the pros and cons by sacrificing some efficiency for broader generalization. So, we would position portfolio mappings between the traditional data structures and learned data structures designed for specific applications.

Accelerating portfolio mapping inference. While portfolio mappings may not currently perform as well as simple machine learning models on today's CPUs (§4), they stand to gain significantly as neural networks—the foundational building blocks of deep learning—continue to be heavily optimized. The rise of AI-optimized hardware, particularly the emergence of AI PCs [7, 8, 18] with features like matrix multiplication instructions and embedded neural network accelerators in CPUs, offers a significant opportunity. Portfolio mappings can leverage these optimizations effortlessly, reducing the performance

gap with specialized learned data structures. Our future work is to port portfolio mappings to those hardware.

Outrageously expensive training. As shown earlier (§4), training portfolio mappings is costly, representing their major limitation. The training process faces two primary bottlenecks: first, the training of individual neural networks, and second, the iterative rounds required to train a sufficient number of neural network candidates to cover the entire input space. Based on our experience, training time increases rapidly with dataset size, leading to scalability challenges. Our future work aims to reduce training time by focusing on simplifying these two phases.

Conclusion. This paper introduces portfolio mappings, a new learned mapping structure that sits between traditional mapping structures and specialized learned models, balancing runtime performance and generalization. We envision that future computer systems will increasingly require more general-purpose, instance-optimized components. Portfolio mappings represent the first step toward realizing this vision.

Acknowledgement

This work was supported by NSF CAREER Award #2237295 and Khoury Seed Grants.

References

[1] Amazon sales rank data for print and kindle books. https://www.kaggle.com/datasets/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.

[2] Wikimedia downloads. http://dumps.wikimedia.org.

[3] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[4] P. Ferragina and G. Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, apr 2020.

[5] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1189–1206, New York, NY, USA, 2019. Association for Computing Machinery.

[6] L. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[7] Q. Incorporated. Unlocking on-device generative ai with an npu and heterogeneous computing. White paper, Qualcomm Incorporated, 2024. Available online: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/Unlocking-on-device-generative-AI-with-an-NPU-and-heterogeneous-computing.pdf.

[8] J.-W. Jang, S. Lee, D. Kim, H. Park, A. S. Ardestani, Y. Choi, C. Kim, Y. Kim, H. Yu, H. Abdel-Aziz, J.-S. Park, H. Lee, D. Lee, M. W. Kim, H. Jung, H. Nam, D. Lim, S. Lee, J.-H. Song, S. Kwon, J. Hassoun, S. Lim, and C. Choi. Sparsity-aware and re-configurable npu architecture for samsung flagship mobile soc. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–28, 2021.

[9] T. Kraska. Towards instance-optimized data systems. *Proceedings of the VLDB Endowment*, 14(12), 2021.

[10] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[11] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2119–2133, New York, NY, USA, 2020. Association for Computing Machinery.

[12] E. Liang, H. Zhu, X. Jin, and I. Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 256–269, New York, NY, USA, 2019. Association for Computing Machinery.

[13] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang. Apex: a high-performance learned index on persistent memory. *Proc. VLDB Endow.*, 15(3):597–610, nov 2021.

[14] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, sep 2020.

[15] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 985–1000, New York, NY, USA, 2020. Association for Computing

Machinery.

[16] A. Rashelbach, O. Rottenstreich, and M. Silberstein. A computational approach to packet classification. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 542–556, New York, NY, USA, 2020. Association for Computing Machinery.

[17] D. Taylor and J. Turner. Classbench: a packet classification benchmark. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2068–2079 vol. 3, 2005.

[18] L. TECHnalysis Research. The ai-pc opportunity. White paper, TECHnalysis Research, LLC, 2023. Available online: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2024-01/the-ai-pc-opportunity-white-paper.pdf.

[19] Y. Wang, C. Tang, Z. Wang, and H. Chen. Sindex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 17–24, New York, NY, USA, 2020. Association for Computing Machinery.