SCIENCE CHINA
Information Sciences

• **Supplementary File** •

A smart hybrid memory scheduling approach using neural models

Yanjie ZHEN[1], Huijun ZHANG[2], Yongheng DENG[1], Weining CHEN[1], Wei GAO[1], Ju REN[1] & Yu CHEN[1]*

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China;*
[2]*China Huaneng Clean Energy Research Institute, Beijing* 000000, *China*

Appendix A Benchmarks

Table A1 Benchmarks statistics.

Benchmark	Suite	Domain	#Unique Pages(4KB)
backprop	Rodinia	Pattern Recognition	42,252
kmeans	Rodinia	Data Mining	34,500
hotspot3D	Rodinia	Physics Simulation	10,733
heartwall	Rodinia	Medical Imaging	9,999
cfd	Rodinia	Fluid Dynamic	8,195
bwaves	SPECCPU2017	Explosion modeling	222,885
wrf	SPECCPU2017	Weather forecasting	49,957
namd	SPECCPU2017	Molecular dynamics	41,157
pennant	CORAL-2	hydrodynamics	67,954
quicksilver	CORAL-2	Monte Carlo	21,543
redis	Real World Application in Linux	NoSQL Databases	104,823
high performance linpack(hpl)	Real World Application in Linux	High-Performance Computing	214,572

Appendix B Limitations of existing page scheduling in hybrid memory systems

Developing effective page scheduling mechanisms is not trivial. Previous research on hybrid memory scheduling can be broadly classified into two categories: non-intelligent scheduling [1–4] and intelligent scheduling [5, 6]. This section provides an overview of these two categories and experimentally demonstrates their limitations.

Appendix B.1 Non-intelligent scheduling

Temporal locality, the tendency of a processor to access the same set of memory locations repeatedly over a short period of time, is a crucial feature of memory access [8]. Many memory management techniques have been designed based on it [9, 10], including most advanced non-intelligent schedulers in hybrid memory systems [1–4]. These schedulers predict the hotness of pages based on the address accesses in the recent past, which are referred to as history schedulers. Nevertheless, history schedulers have limited prediction capabilities due to the short length of the address sequence they look back on, which makes them less robust for applications with sharply changing access patterns or frequent random accesses.

To investigate the limitations of history schedulers, we use ten benchmarks from Rodinia [11], SPECCPU2017 [12], CORAL-2 [13] and two real-world applications, which cover diverse domains and are summarized in Table A1. Following several state-of-the-art schedulers, we implement a history scheduler that assumes pages hot in the current epoch will also be hot in the next epoch. Then, we simulate a hybrid memory system consisting of fast and slow memory with a capacity ratio of 1:8 and deploy the history scheduler to the simulated system. Besides, we also implement an oracle scheduler [3] that assumes all future page accesses are already known, which serves as the performance upper bound for comparison. We use the hit rate of fast memory as the performance evaluation metric of schedulers, and the evaluation results are shown in Figure B1. We can observe that the fast memory hit rate of the history scheduler is 37.9% on average, which lags behind 88.9% of the oracle scheduler. Therefore, new page scheduling policies are in dire need of bridging the significant performance gap.

* Corresponding author (email: yuchen@tsinghua.edu.cn)

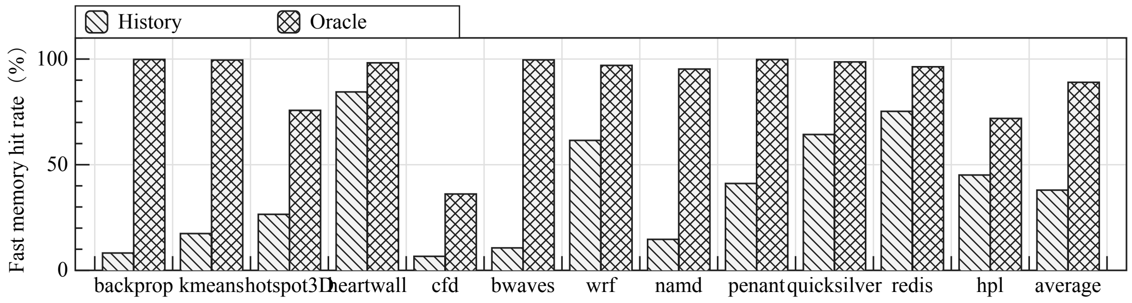

Figure B1 The performance gap between the history scheduler and the oracle scheduler. The oracle scheduler is the optimal scheduler that assumes all future page accesses are already known. Section Appendix D.1 details the experimental methodology.

Appendix B.2 Intelligent scheduling

An application's memory access behavior is determined by how its code executes, resulting in predictable access patterns. However, the memory access patterns can be quite complex and vary depending on the application's specific function. Solely relying on temporal locality is insufficient to capture this complexity accurately. Effective scheduling should possess the capability to identify complex memory access patterns and make well-informed predictions based on these patterns. Inspired by the ability of deep learning to learn complex patterns and its competitive performance in computer architecture, especially in hardware prefetching [14–17], some works have investigated the potential of using neural models in hybrid memory scheduling.

To the best of our knowledge, Kleio [5] and Coeus [6] are the only two hybrid memory page schedulers employing neural models. These works have demonstrated that neural models can significantly improve the accuracy of page hotness prediction. However, their exorbitant costs impede their wide use in practice since both of them utilize individual prediction, i.e., deploying a Recurrent Neural Network (RNN) for each page. Such a design comes with a prohibitive cost, which increases proportionally with the number of pages. Each application has a large number of unique pages, with the max exceeding 220,000, as shown in Table A1. Concurrently training and inferring such a large number of RNN models can be costly in terms of both time and memory, rendering it impractical use. Kleio and Coeus propose to limit the number of neural models to reduce prediction cost. They divide pages into two categories: important pages are predicted by RNN models, while the remaining pages are predicted by traditional history scheduling. However, we have experimentally demonstrated that employing such a design cannot reduce the cost of these schedulers to a practical level.

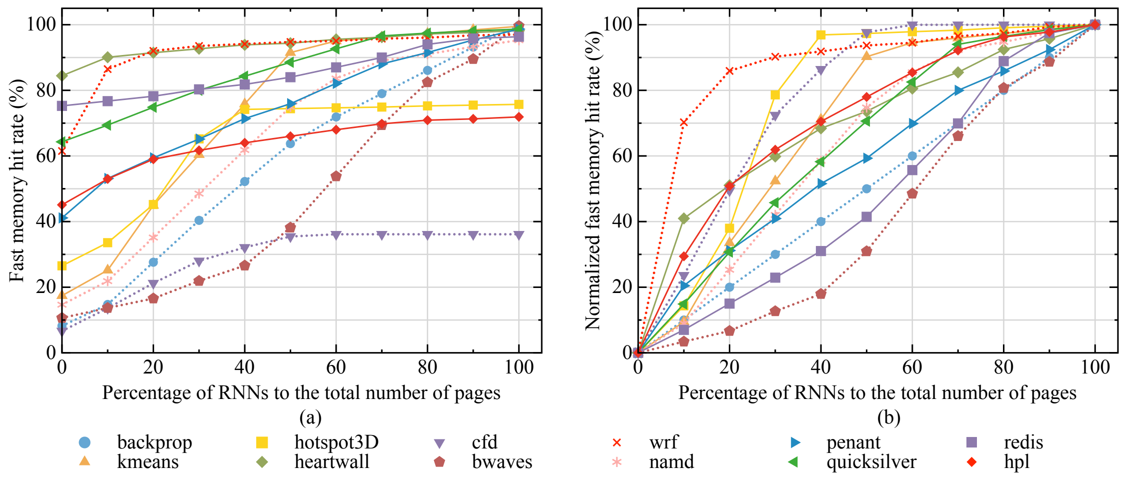

Figure B2 The fast memory hit rate when tuning the percentage of RNNs to the total number of pages.

To evaluate the impact of the number of RNNs (corresponding to the number of pages predicted using neural models) on system performance, following the settings in Kleio, we select a subset of pages and deploy an RNN for each, while the remaining pages are still managed by a history scheduler. Figure B2(a) illustrates the hit rate of the fast memory when tuning the percentage of RNNs to the total number of pages. We can observe that when the percentage of RNNs is set to 0, all pages are managed by the history scheduler, and when set to 1, all pages are managed by RNNs. We employ the oracle scheduler to predict the selected pages rather than using RNNs to avoid the high cost of numerous RNNs. This simplification does not affect the conclusion, as the oracle scheduler is the upper limit of RNNs. Figure B2(b) normalizes the hit rate improvement. We observe that when the hit rate improvement achieved by RNNs is half of the maximum possible improvement, the number of pages managed by RNNs ranges from 7.7% to 60.8% of the total pages, averaging 36.4%. Referring to Table A1, for each benchmark, RNNs are required to manage an average of approximately 18,000 pages, with a maximum of 135,500 pages. Given that each page needs to build an RNN model, the total number of required models is significantly high, making the neural-model-based scheduler impractical.

The above experimental results reveal that the cost of existing neural-model-based hybrid memory scheduling schemes is too expensive to be practical. Consequently, reducing the cost of the neural-model-based scheduling scheme is essential to promote its wide use in practice.

Appendix C Cost-driven approach: from individual to collective

This section explores converting the hybrid memory scheduling problem into a neural network prediction task. A critical aspect of employing neural networks is the selection of features that accurately represent the problem and serve as inputs. We discuss the choice of input features with a focus on cost considerations.

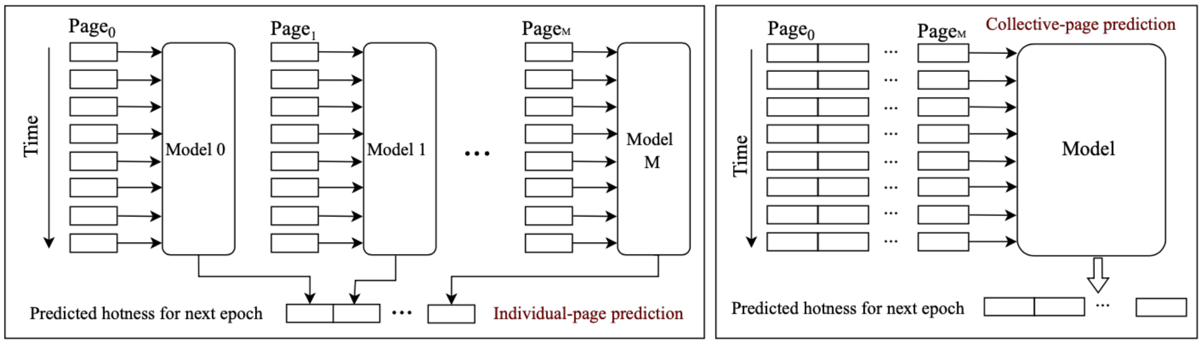
Page$_0$
Page$_1$
Page$_M$
Time
Model 0
Model 1
Model M
Predicted hotness for next epoch
Individual-page prediction
Page$_0$
Page$_M$
Collective-page prediction
Time
Model
Predicted hotness for next epoch

Figure C1 Input features of individual-page prediction and collective-page prediction.

Appendix C.1 Individual-page prediction

Existing neural-model-based schedulers employ a separate neural model for each page, which is referred to as individual-page prediction. Figure C1 illustrates the input of individual-page prediction. Despite the high prediction accuracy of each model, the significant cost of deploying a large number of models makes these schedulers impractical for real-world systems. We experimentally measure the cost of individual-page prediction models with a comprehensive experimental setup detailed in Section Appendix D. The cost can be categorized into two areas:

1. Time and memory for training. During training, memory usage for each model can be up to tens of gigabytes(GB). However, these models cannot be trained in parallel even when using advanced GPUs, such as the NVIDIA A100 Tensor Core GPU [18]. Specifically, it takes approximately ten minutes to train each model. As discussed in Section Appendix B, prior works relying on individual-page prediction require thousands or even hundreds of thousands of models. Training such a large number of models serially consumes significant hardware resources and time.

2. Time and memory for inference. The duration of a scheduling epoch in most current hybrid memory schedulers typically spans from 1 to 10 seconds [3, 19–21], which provides a baseline for our cost analysis. The neural model's inference time must be shorter than a single epoch while ensuring ample time for page migration. Our experiments show that the resource utilization for one model in individual-page prediction during inference occurs almost instantaneously and takes approximately 80 milliseconds on average. However, it is worth noting that each model requires a separate process or thread for inference. As the number of models increases, this can consume substantial scheduling and thread resources, resulting in an extended time required for completing the inference of all models.

Appendix C.2 Collective-page prediction

In fact, neural models have the ability to learn complex patterns, such as learning different access patterns of multiple pages simultaneously. To exploit the full potential of neural models, we propose to deploy a single model for all pages (collective-page prediction) instead of deploying an individual neural model for each page (individual-page prediction). Figure C1 exhibits the input features of the collective-page prediction model, which feeds the hotness of all pages within an epoch as a whole to a single neural model. In contrast with individual-page prediction, collective-page prediction has the following significant advantages:

1. Reducing the number of models, thus avoiding the cost of thread resources and scheduling.

2. Reducing the parameters of models, thus reducing computational and memory costs. While the single model employed for collective-page prediction requires more parameters to capture complex patterns than each model in individual-page prediction, the relationship between the number of parameters in a neural model and the range of page patterns it covers is non-linear. Consequently, the total number of parameters required for collective-page prediction is considerably less than that required for individual-page predictions.

3. More suited for existing hardware. Collective-page prediction can utilize hardware resources more efficiently by leveraging the simultaneous computation of numerous parameters per network layer and batch processing of multiple pages in a large model because current deep learning-supporting hardware possesses robust matrix computation capabilities.

Appendix D Experiment

This section evaluates the performance of SmartS by comparing it with both existing non-intelligent schedulers and neural-model-based schedulers.

Appendix D.1 Methodology

Neural network models are typically trained offline on an input corpus. However, we conducted a simulated online training experiment similar to [22] to validate that SmartS can capture changes in memory access patterns during program execution. Specifically, SmartS is trained on a phase of 50 million memory accesses and uses the trained model to predict the next phase of 50 million memory accesses. Thus, the model is constantly being trained during one phase for use in the next phase. No inference is performed in the first phase.

Simulator. We developed a Python-based simulation of a hybrid memory system similar to [3, 5], whose accuracy has been validated by its developers through comparison with performance data from real hybrid memory systems. The simulated hybrid memory consists of a fast and a slow memory component. Both components are managed uniformly as contiguous physical memory, similar to the app direct mode configuration of the Intel Optane platform [7]. The overall capacity of the simulated memory system is equivalent to the memory footprint of the application, with a fast to slow memory capacity ratio of 1:8. Based on the PMEM access speeds reported in [3, 7], we established a latency ratio of 1:3 and a bandwidth ratio of 1:0.37 between the fast and slow memory components. To obtain an accurate estimation of the runtime, we add constant delays for every page migration and the start of a period to account for the cost of the page scheduler itself, using the proposed values in [23]. Furthermore, we assumed that an epoch corresponds to the time required to issue a fixed number of memory accesses since we do not employ cycle-accurate simulation. In our experiments, we assume 10,000 memory access requests per epoch, similar to [20].

Datasets. We selected ten benchmarks from Rodinia [11], SPECCPU2017 [12], and CORAL-2 [13] to evaluate the performance of SmartS. These benchmarks cover various application scenarios with varying memory access patterns. To evaluate SmartS on more challenging workloads, we also use two real-world applications: Redis, a popular NoSQL database, and High Performance Linpack (HPL), a software application designed for assessing the performance of high-performance computing systems.

In line with prior works [3, 5], we utilize the frequency of page accesses within a given epoch as the page's hotness during this epoch. We employ Intel's Pin [24] dynamic binary instrumentation tool to capture the memory access trace and tally the page access frequencies for each epoch. Since we focus on accesses to memory, we add a simulated three-level cache when collecting accessed addresses. The final trace collected is the memory address of the last level cache misses.

Baseline. We evaluate SmartS against the history scheduler [3], Kleio [5], Coeus [6] and the oracle scheduler [3]. The history scheduler is a state-of-the-art non-intelligent scheduler that uses the current epoch's page hotness as the next epoch's page hotness. Kleio and Coeus are state-of-the-art neural-model-based schedulers. We implement them following the settings in their papers. Both deploy 100 RNNs for selected pages (100 pages in Kleio and 100 identical patterns in Coeus) and manage the remaining pages using a history scheduler. The oracle scheduler is an ideal but impractical scheduler that assumes the future address access is known. It is the upper limit of all schedulers.

Appendix D.2 Cost analysis

First and foremost, we evaluate the cost of different schedulers. We conduct experiments using a machine with a dual-socket 3.00GHz Intel Xeon Gold 6248R processor and 256GB of memory. SmartS's neural model is trained offline and accelerated using an A100 GPU with 40GB of memory. We report the average metrics obtained from the given hardware testbed. The hyperparameters of the model are determined based on the number of clusters. We present the cost associated with a configuration of 2,000 clusters, as Figure D3 indicates that the performance of most benchmarks converges when the number of clusters reaches 2,000. Consequently, the hyperparameters of the model, embedding length and hidden length, are both set to 2,000. We also deployed Kleio and Coeus in the same environment, with code obtained from their open-source repositories.

The training cost of SmartS consists of two parts: one is the cost of clustering, and the other is the cost of training the LSTM model. The clustering time increases with the number of pages, ranging from 11 to 313 seconds, averaging around 121 seconds. The time of training the LSTM model depends on the number of clusters. When the number of clusters is set to 2000, the training time of SmartS spanned an average of 80 epochs, lasting approximately 3 hours. Memory utilization during training peaked at tens of gigabytes. The memory usage for saving the trained model is less than 40MB. Regarding resource usage during inference, it is trivial and takes an average of only 6.8 milliseconds without GPU acceleration and 1.3 milliseconds with GPU acceleration. The typical duration of a hybrid memory scheduling epoch ranges from 1 to 10 seconds [3, 19–21]. SmartS' inference time is acceptable and leaves ample time for page migration.

For comparison, we also evaluate the cost of each model in Kleio and Coeus. Their training phase takes an average of 10 minutes for each model, and memory utilization during training for each model is also at the gigabyte level. The memory usage for saving the trained model is 1.5 MB and the average inference time is approximately 3 milliseconds without GPU acceleration. However, Kleio and Coeus require building thousands or even tens of thousands of neural models for a single application, as discussed in Section Appendix B. Although the cost of each individual model is slightly lower than that of SmartS, the cumulative cost of numerous models, along with the scheduling cost, becomes impractical.

In conclusion, compared to prior neural-model-based scheduling, SmartS's clustering-based similar page identifier and LSTM-based collective-page prediction significantly reduce memory and computation costs.

Appendix D.3 Comparing effectiveness with prior art

Appendix D.3.1 *Prediction accuracy of neural models*

Prior works have explored using neural models to predict page hotness. Their empirical results have demonstrated that neural models can achieve high prediction accuracy. However, previous approaches build one model for each identical pattern corresponding to one or multiple pages. Conversely, SmartS deploys a single model to learn numerous different patterns corresponding to all pages. SmartS utilizes fewer models to learn more patterns compared to prior approaches. In this section, we experimentally answer whether SmartS can maintain high prediction accuracy while reducing the number of models and increasing the number of learned patterns.

Selecting an appropriate metric to measure the prediction accuracy is a crucial step in our analysis. SmartS and prior works' models utilize regression predictions to estimate page hotness. Common metrics such as Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and R-squared (R^2) are widely used to calculate the distance between the predicted values and true values to measure the accuracy of regression predictions. However, they may not be ideal for our purposes. This is because these models' primary goal is not to predict the pages' hotness accurately but to identify the most promising candidates for migration based on their relative hotness rankings. We adopted fast memory hit rate as an alternative metric to evaluate whether the model correctly selected higher hot pages for migration. The preemptive migration of pages into the fast memory before their access can enhance the fast memory hit rate. The higher the hotness of the migrated pages, the more significant the improvement on the fast memory hit rate.

Notably, Kleio and Coeus' neural models predict a different range of pages compared to SmartS. Kleio and Coeus' model can only predict the hotness of a limited number of selected pages, whereas SmartS' model can predict the hotness of all pages.

For a fair comparison, we calculate the hit rate by excluding pages that do not employ neural model predictions, thus centering attention exclusively on the improvement of the hit rate that the neural models can furnish on selected pages. Specifically, in Kleio and Coeus, fast memory hit rate improvement is normalized between 0% and 100%, where 0% represents all pages are managed by the history scheduler, and 100% represents the selected pages are managed by the oracle scheduler and the remaining pages are managed by a history scheduler. Following the settings described in their paper, Kleio and Coeus built 100 LSTM models, where Kleio intelligently predicted 100 pages, and Coeus intelligently predicted an average of 390 pages. In SmartS, fast memory hit rate improvement is similarly normalized, with 0% representing all pages are managed by the history scheduler and 100% representing all pages are managed by the oracle scheduler.

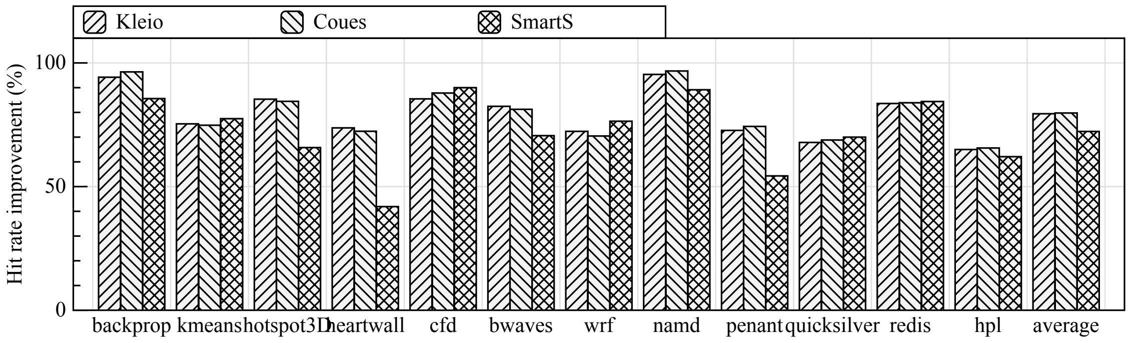

Figure D1 Normalized fast memory hit rate improvement of Kleio, Coeus and SmartS.

Figure D1 illustrates the fast memory hit rate improvement achieved by SmartS compared to Kleio and Coeus. SmartS's number of clusters is set to 5000 as a trade-off between performance and cost. The effect of the number of clusters on SmartS will be discussed in Section Appendix D.4.1. On average, SmartS brings 72.3% of the possible fast memory hit rate improvement, while Kleio and Coeus bring 79.5% and 79.8%. For 66%applications, SmartS yields a comparatively lower hit rate improvement than Kleio and Coeus. However, the gap in hit rate improvement between these schedulers is relatively small, with a variation of less than 10% in 66% of applications.

SmartS's hit rate improvement is lower than prior works for three primary reasons. Firstly, SmartS uses a single model to learn multiple patterns simultaneously, while Kleio and Coeus use a single model to learn a single pattern. Secondly, SmartS predicts a more significant number of pages than Kleio and Coeus. SmartS incorporates a clustering mechanism to effectively predict a large number of pages while accounting for cost constraints, introducing partial errors. Lastly, SmartS predicts all pages, unlike Kleio and Coeus, which selectively predict specific pages through the page selector.

In summary, despite the reduction in the number of models and the increase in the scope of learned patterns, the neural model in SmartS maintains high prediction accuracy, with only a decrease of 8.7% compared with prior schedulers' neural models. It is worth noting that the performance improvement gained from a single page is limited. SmartS's neural model manages a number of pages that is tens or even thousands of times greater than prior works. As shown in Section Appendix B, only a scheduler like SmartS, capable of intelligently predicting numerous pages, can significantly improve the hybrid memory systems' performance.

Appendix D.3.2 *Application performance*

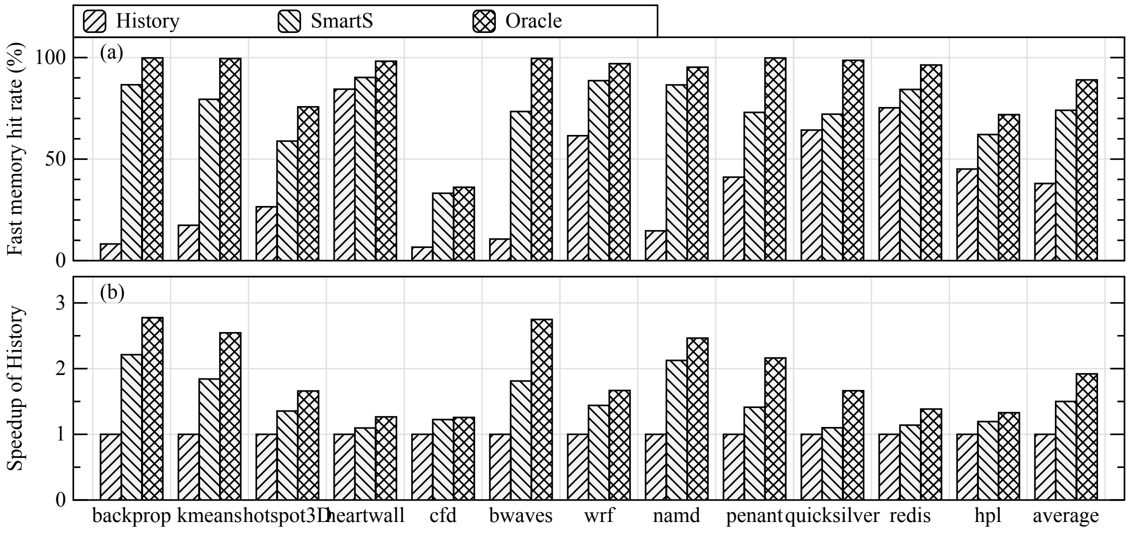

Figure D2 Fast memory hit rate and speedup in the application runtime of Kleio, Coeus and SmartS.

Apart from the model's prediction accuracy, the improvement of system performance is a more important metric to evaluate the hybrid memory schedulers. In this section, we evaluate the fast memory hit rate of SmartS compared to the history scheduler and the oracle scheduler on ten applications. Furthermore, we evaluate the speedup in the application runtime of SmartS and the oracle scheduler, with the history scheduler as the baseline and being normalized to 1. To calculate the speedup of various schedulers, we employ the analytical model utilized by Meswani et al. [3] to estimate the application runtime based on the number of accesses serviced from fast memory and slow memory appropriately. This model uses the Leading Loads method, which divides the application runtime into computation time and the time required to satisfy memory requests. A comparison between SmartS, Kleio, and Coeus is unnecessary since Section Appendix B clearly shows the relationship between the number of models and performance. Specifically, the performance improvement is relatively insignificant when utilizing a limited number of models, such as the 100 models used in Kleio.

SmartS improves the hit rate on average to 74.0%, as shown in Figure D2(a). SmartS brings 70.5% of the possible hit rate improvement, compared with 37.9% for the history scheduler and 89.1% for the oracle scheduler. Figure D2(b) displays the speedup in the application runtime achieved by SmartS, which is 1.50x compared with 1.92x of the oracle scheduler. SmartS has significant advantages for most applications compared with the history scheduler, especially for scientific computations and big data analytics applications, such as *backprop*, *kmeans*, *namd*, etc. For these applications, there is a significant gap between the history scheduler and the oracle scheduler, while SmartS effectively bridges this gap. The reason is that such applications always iterate forward, similarly traversing the addresses during each iteration cycle. There are multiple scheduling epochs within each iteration cycle. Scheduling epochs within the same iteration cycle traverse different memory regions. Consequently, the history scheduler that predicts based on the last one or a few scheduling epochs performs poorly, while SmartS that predicts based on a long history from several iteration cycles significantly improves performance. Even for benchmarks where the history scheduler performs well, SmartS still performs better, e.g., the hit rate of *heartwall*, *wrf*, and *quicksilver* can achieve up to 76.44% of the possible hit rate improvement.

Appendix D.4 Understand clustering in SmartS

This section analyzes our results to illustrate the effectiveness of the clustering-based similar page identification mechanism in SmartS. We pay particular attention to (1) how to set the number of clusters and (2) the performance reduction caused by clustering.

Appendix D.4.1 *Effect of the number of clusters*

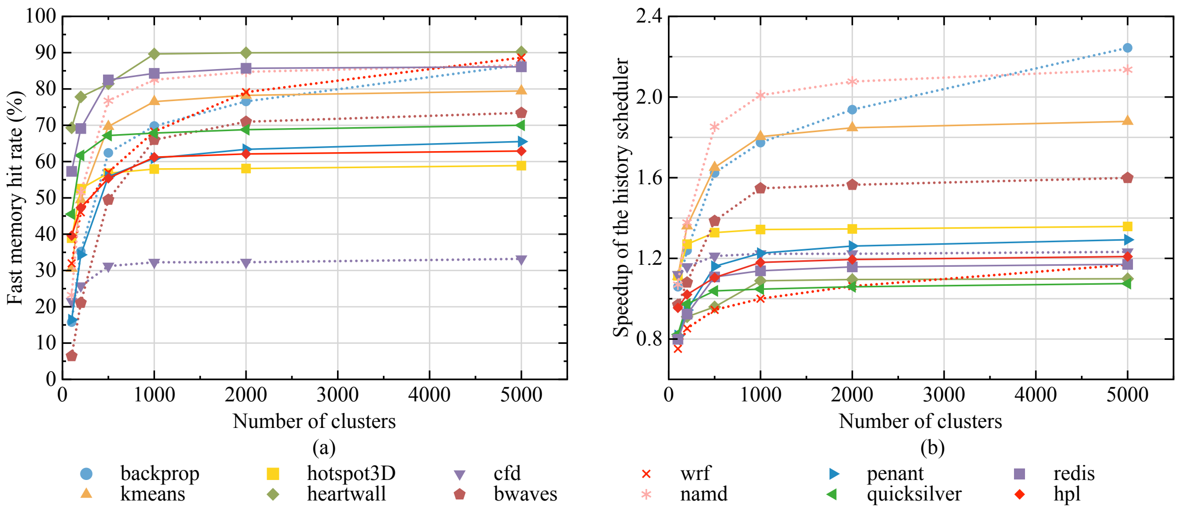

Figure D3 Fast memory hit rate and speedup in the application runtime of SmartS with different number of clusters.

This section mainly discusses the effect of the number of clusters on model effectiveness and provides recommendations on selecting the optimal cluster number. Figure D3 illustrates the fast memory hit rate and speedup of SmartS when the number of clusters is 100, 200, 500, 1000, 2000, and 5000. It can be observed that as the number of clusters increases, the effectiveness of SmartS progressively improves. This is due to clusters grouping pages with similar patterns. The hotness of a page is replaced by the average hotness of pages within that cluster during the hotness prediction phase. The discrepancy between the true hotness of a page and the average hotness is the error introduced by clustering. Naturally, with fewer clusters, each cluster encompasses more pages, leading to a larger error.

We observe that each curve exhibits an elbow point. In other words, SmartS is only sensitive to the number of clusters when it is below the elbow point. When the cluster number exceeds the elbow point, SmartS's sensitivity to the number significantly diminishes. This suggests that the similarity among pages belonging to the same cluster has become considerably high. Based on this observation, we recommend selecting the elbow point where the performance curve begins to plateau as the cluster number. From our experiments, 1000-2000 is a suitable inflection point for most applications.

The elbow point also indicates that for different applications with a page number ranging from a few thousand to several hundred thousand, only 1,000 to 2,000 clusters in SmartS are needed to achieve high effectiveness. The reason is that although there are many pages, the number of different access patterns is limited. Therefore, SmartS is capable of accommodating variations in the number of pages.

Appendix D.4.2 *Breakdown of performance reduced by clustering*

Some pages may migrate incorrectly due to clustering. The decision to migrate pages is based on the average hotness of the pages in the cluster to which they belong. However, some pages significantly deviate from this average, suggesting they should be treated differently.

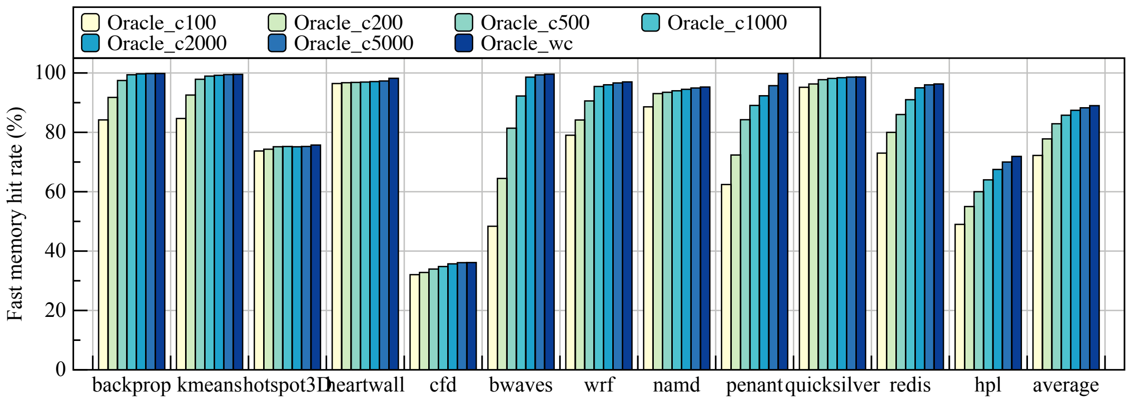

Figure D4 Fast memory hit rate of the oracle scheduler with different numbers of clusters.

We employ the same clustering strategy as SmartS on the oracle scheduler to break down the negative impact of the clustering on SmartS. Figure D4 illustrates the fast memory hit rate of the clustering-based oracle scheduler with the number of clusters of 100(*Oracle_c100*), 200(*Oracle_c200*), 500(*Oracle_c500*), 1000(*Oracle_c1000*), 2000(*Oracle_c2000*) and 5000 (*Oracle_c5000*), compared with the oracle scheduler without clustering(*Oracle_wc*). The result shows that, on average, the oracle scheduler with 1000, 2000 and 5000 clusters reduces the hit rate to 85.8%, 87.4% and 88.3%, compared with 88.9% for the oracle scheduler without clustering. It demonstrates that our clustering strategy only has a minor negative impact while effectively reducing the input and output size of the neural model.

However, another doubt might emerge if we compare the two experiments about the number of clusters followed by SmartS and the oracle scheduler. That is why SmartS is more sensitive to the change in the number of clusters, while the oracle scheduler is less affected. The reason could be traced back to the characteristics of neural networks. The auto-learning ability of neural networks is based on the learning of the given input, which means that the more information is given as input, the more effective information related to the task could be obtained. The number of clusters directly corresponds to the length of the input features, thus directly affecting the information quantity. Therefore, when the number of clusters is lower than 500, the performance of SmartS sharply declines, ascribing to too little information.

Appendix D.5 Validation of predicted k value

This section aims to evaluate whether the k value has a pattern, whether the pattern is learnable, and the impact of the errors introduced by the predicted k values on the effectiveness of SmartS. Metric MAPE (Mean Absolute Percentage Error) is employed to evaluate the accuracy of the k value regression model:

$$MAPE(K,\tilde{K}) = \frac{100\%}{N}\sum_{i=1}^{N}\left|\frac{K_i-\tilde{K}_i}{K_i}\right| \tag{D1}$$

where K_i is the true value calculated from page hotness and $\tilde{K}_i$ is the predicted value. MAPE is a very intuitive interpretation in terms of relative error. As shown in Figure D5(a), the MAPE ranges from 0.002% to 18.0% on 12 benchmarks, with 10 of them less than 10%. This confirms the existence of patterns of k value and shows how regression prediction models can learn the pattern.

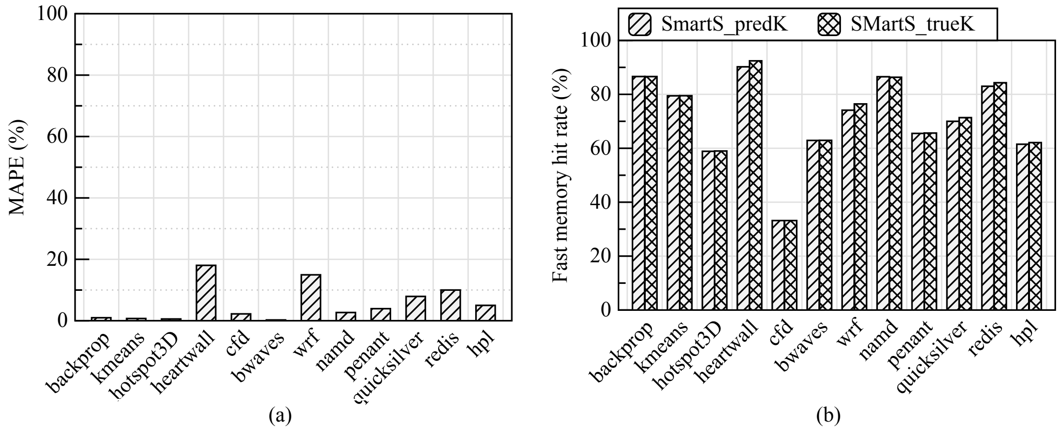

Figure D5 MAPE of predicted k value and decrease in SmartS' hit rate caused by the error of predicted k values.

Figure D5(b) depicts the decrease in SmartS' hit rate caused by the error of predicted k values. Even for *wrf* and *heartwall* with the MAPE exceeds 10%, the hit rate fall is less than 2%, which is substantially less than the MAPE. The reason is that SmartS has ordered the pages according to page hotness. The pages that migrate incorrectly fall in the pages with low hotness at the end of the pages queue, considerably diminishing the little impact on scheduler performance.

Appendix E Discussion

In this section, we will discuss the limitations of the current study and potential directions for future research.

Although SmartS significantly reduces the cost of training and inference compared to previous works, the training time is still too long for applications with frequently changing memory access patterns and high real-time requirements. This is also an issue that researchers are actively working to address, but currently, there is no effective solution.

There are some possible paths that might address the issue of long training time. Transfer learning approaches [28] could be considered whereby a foundational model is pre-trained and then fine-tuned based on new memory access patterns. Incremental learning [27] also could be explored, allowing the model to be updated incrementally based on emerging memory access patterns. Furthermore, model compression techniques, such as knowledge distillation [26] and weight pruning [25], can be considered to reduce the model's size and computational demands, making it more suitable for online updates.

Integrating neural models into real-world operating systems is another important research direction for the future and poses multiple challenges: (1) Collecting and storing page access information at a low cost. Page access information serves as the data input for model training. Collecting and storing this information in a lightweight approach requires further study. (2) Reducing the resource consumption for model training and inference. Model training and inference can impact the resource scheduling of the operating system and the execution speed of applications. Therefore, minimizing the resource consumption, including both computational and storage resources, for model training and inference is necessary.

References

1 Yang D, Liu H, Jin H, et al. Hmvisor: dynamic hybrid memory management for virtual machines. Sci China Inf Sci, 2021, 64(9): 192104

2 Shen D, Liu X, Lin FX. Characterizing emerging heterogeneous memory. ACM Sigplan Not, 2016, 51(11): 13-23

3 Meswani MR, Blagodurov S, Roberts D, et al. Heterogeneous memory architectures: a HW/SW approach for mixing die-stacked and off-package memories. In: Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA), 2015. 126-136

4 Wu K, Huang Y, Li D. Unimem: runtime data management on non-volatile memory-based heterogeneous main memory. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017. 1-14

5 Doudali TD, Blagodurov S, Vishnu A, et al. Kleio: a hybrid memory page scheduler with machine intelligence. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, 2019. 37-48

6 Doudali TD, Gavrilovska A. Coeus: clustering (a) like patterns for practical machine intelligent hybrid memory management. In: Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, 2022. 615-624

7 Izraelevitz J, Yang J, Zhang L, et al. Basic performance measurements of the Intel Optane DC persistent memory module. 2019. ArXiv:1903.05714

8 Stallings W. Computer Organization and Architecture: Designing for Performance. Pearson Education India, 2003.

9 Traverso S, Ahmed M, Garetto M, et al. Temporal locality in today's content caching: why it matters and how to model it. ACM SIGCOMM Comp Commun Rev, 2013, 43(5): 5-12

10 Jaleel A, Borch E, Bhandaru M, et al. Achieving non-inclusive cache performance with inclusive caches: temporal locality aware (TLA) cache management policies. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010. 151-162

11 Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009. 44-54

12 Bucek J, Lange K-D, v. Kistowski J. SPEC CPU2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, 2018. 41-42

13 Wu X. and Taylor V. Power and Performance Characteristics of CORAL Scalable Science Benchmarks on BlueGene/Q Mira. In: Proceedings of the International Green and Sustainable Computing Conference (IGSC), 2015, 1-6

14 Hashemi M, Swersky K, Smith J, et al. Learning memory access patterns. In: International Conference on Machine Learning, 2018, PMLR. 1919-1928

15 Zeng Y, and Guo X. Long Short Term Memory Based Hardware Prefetcher: A Case Study. In: Proceedings of the International Symposium on Memory Systems, 2017. 305-311

16 Narayanan A, Verma S, Ramadan E, et al. DeepCache: A deep learning based framework for content caching. In: Proceedings of the 2018 Workshop on Network Meets AI & ML, 2018. 48-53

17 Peled L, Weiser U, Etsion Y. A Neural Network Prefetcher for Arbitrary Memory Access Patterns. ACM Trans Archit Code Optim, 2019, 16(4): 1-27.

18 Choquette J, Gandhi W, Giroux O, et al. NVIDIA A100 tensor core GPU: performance and innovation. IEEE Micro, 2021, 41(2): 29-35

19 Yan Z, Lustig D, Nellans D, et al. Nimble page management for tiered memory systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 331-345

20 Agarwal N, Wenisch T F. Thermostat: application-transparent page management for two-tiered main memory. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017. 631-644

21 Kwon Y, Yu H, Peter S, et al. Coordinated and Efficient Huge Page Management with Ingens. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. 705-721

22 Shi Z, Jain A, Swersky K, et al. A hierarchical neural model of data prefetching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 861-873

23 Kommareddy V R, Hammond S D, Hughes C, et al. Page migration support for disaggregated non-volatile memories. In: Proceedings of the International Symposium on Memory Systems, 2019. 417-427

24 Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Not, 2005, 40(6): 190-200

25 Hassibi B, Stork D. Second order derivatives for network pruning: Optimal brain surgeon[J]. Advances in neural information processing systems, 1992, 5.

26 Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network[J]. arXiv preprint arXiv:1503.02531, 2015.

27 Parisi G I, Kemker R, Part J L, et al. Continual lifelong learning with neural networks: A review[J]. Neural networks, 2019, 113: 54-71.

28 Pan S J, Tsang I W, Kwok J T, et al. Domain adaptation via transfer component analysis[J]. IEEE transactions on neural networks, 2010, 22(2): 199-210.