Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence

Thaleia Dimitra Doudali
Georgia Institute of Technology
thdoudali@gatech.edu

Sergey Blagodurov
Advanced Micro Devices, Inc.
Sergey.Blagodurov@amd.com

Abhinav Vishnu
Advanced Micro Devices, Inc.
Abhinav.Vishnu@amd.com

Sudhanva Gurumurthi
Advanced Micro Devices, Inc.
Sudhanva.Gurumurthi@amd.com

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

ABSTRACT

The increasing demand of big data analytics for more main memory capacity in datacenters and exascale computing environments is driving the integration of heterogeneous memory technologies. The new technologies exhibit vastly greater differences in access latencies, bandwidth and capacity compared to the traditional NUMA systems. Leveraging this heterogeneity while also delivering application performance enhancements requires intelligent data placement. We present **Kleio**, a page scheduler with machine intelligence for applications that execute across hybrid memory components. Kleio is a hybrid page scheduler that combines existing, lightweight, history-based data tiering methods for hybrid memory, with novel intelligent placement decisions based on deep neural networks. We contribute new understanding toward the scope of benefits that can be achieved by using intelligent page scheduling in comparison to existing history-based approaches, and towards the choice of the deep learning algorithms and their parameters that are effective for this problem space. Kleio incorporates a new method for prioritizing pages that leads to highest performance boost, while limiting the resulting system resource overheads. Our performance evaluation indicates that Kleio reduces on average 80% of the performance gap between the existing solutions and an oracle with knowledge of future access pattern. Kleio provides hybrid memory systems with fast and effective neural network training and prediction accuracy levels, which bring significant application performance improvements with limited resource overheads, so as to lay the grounds for its practical integration in future systems.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning approaches**; • **Hardware → Memory and dense storage**; **Analysis and design of emerging devices and systems**; • **General and reference → Performance**;

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325398

KEYWORDS

Data Tiering; Emerging Memory Technologies; Heterogeneous Memory Systems; Hybrid Memory Systems; Long Short Term Memory Networks; Machine Intelligence; Machine Learning; Non Volatile Memory; Page Scheduler; Recurrent Neural Networks;

ACM Reference Format:
Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325398

1 INTRODUCTION

Modern systems are frequently designed using heterogeneous memory components. These memories are typically leveraged for extending main memory capacity or for caching purposes. There are natural trade-offs in the hybrid memory systems (HMS) comprising heterogeneous components. Typically deeper memory (further from the compute unit (CPU/GPU) has more capacity albeit at larger latency and reduced bandwidth.

We consider one such HMS scenario comprising of DRAM and Non Volatile Memory (NVM) and focus on the problem of extending main memory capacity. An important artifact of HMS is addressing the limitations of increased latency and decreased bandwidth with deeper memories. In our case, a **page scheduler** – the memory management layer of operating and runtime systems – is responsible for the page migration across the heterogeneous memory components. An effective page scheduler is responsible for ensuring that *hot pages – the ones that are accessed frequently* – are readily available in faster memory (DRAM). This is an intricate task, especially it is a complex combination of access pattern of pages in an application, and its runtime parameters (input size, strong/weak scaling, etc.). To address this challenge, several researchers have considered solutions whose implementation can be integrated in the hardware-, compiler-, Operating System-, runtime-, hypervisor- or application profiling-level [7, 9, 11, 15, 20, 27–29]. A common theme among these approaches is that they rely exclusively on historic information about page accesses. Specifically, the state-of-the-art [20, 27, 28] in system-level dynamic page management solutions for HMS utilize the immediate observed behavior to make decisions on the best future page placement. However, as we show in this paper, the mispredictions regarding future page access resulting from use of historic information alone, can leave an up to

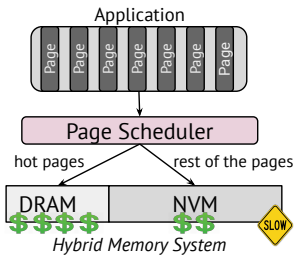


Figure 1: The high cost of DRAM in large capacities limits its future use and makes a case for denser but cheaper technologies (e.g., NVM) to be used in extending the system's memory capacity. In such a hybrid system the page scheduler periodically migrates pages such that the ones that are frequently accessed get allocated in the fastest available memory technology (e.g., DRAM) until the capacity is full.

55% gap in the obtained versus attainable application performance, thus failing to fully leverage the aggregate HMS resources.

An Operating System (OS) level memory management solution may be implemented inside the OS kernel's memory manager, or on the user level with OS system calls for page migration, similar to Linux `move_pages()`. In this potential implementation, the current state-of-the-art **History page scheduler** [20] would periodically migrate pages, such that those that are hot for the current scheduling epoch, get allocated in DRAM until capacity is full, with the hope to be frequently accessed during the next scheduling epoch (Figure 1).

Although the History page scheduler is relatively straightforward and practical to implement, its effectiveness in providing applications with fast (i.e., in-DRAM) data accesses inherently depends on the application data access behavior. When comparing with an **Oracle page scheduler**, which uses a-priori knowledge to periodically migrate application pages such that those that are indeed highly accessed in the next scheduling epoch (hot pages) get allocated in DRAM until capacity is full, we observe that a history-based page scheduler will result in significant reduction in fast memory accesses and subsequent application slowdown (Section 2). The exact impact depends on application data access behavior and the capacities and performance characteristics of the different memories. This illustrates an important point: *Purely history-based page scheduling methods are limited in the performance opportunities they can provide to applications running on hybrid memory systems. Instead, they must be augmented with more intelligent, predictive methods.*

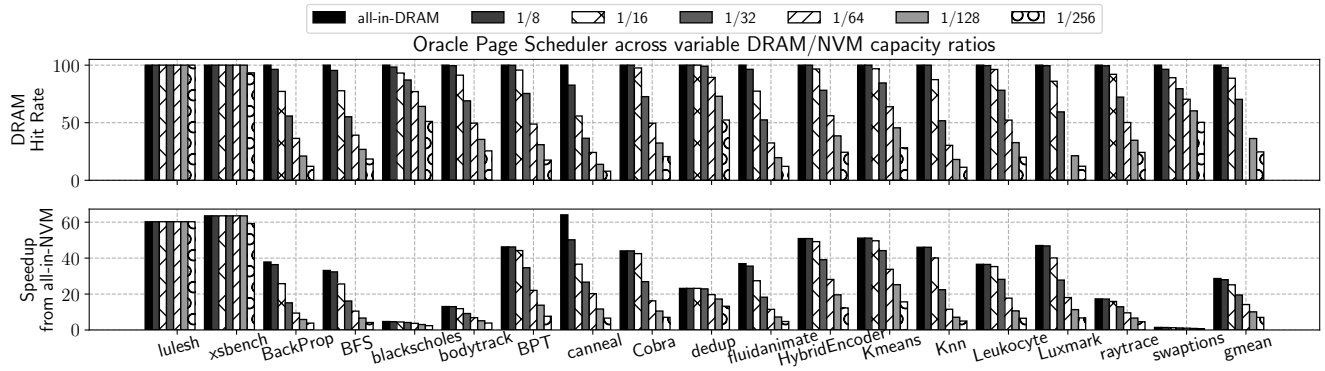
Why a solution with Machine Intelligence (MI)?
As shown in Section 2, the immediately observed memory access behavior is insufficient in capturing the necessary information that predicts future behavior for making clever placement decisions. Yet, a larger window of accesses should allow the ability to capture the historic information (*long term access*, and also leverage the recent accesses (*short term access*) for effective page placement. There are a few design possibilities: 1) Use simple methods such as Markov chains for handling the temporal aspect [25], 2) use advanced techniques with machine intelligence that provide mechanisms to handle temporal data capturing both short and long term page access patterns. Such techniques are reinforcement learning and deep neural networks (recurrent neural / long short term memory networks), which are currently widely explored to solve various systems problems as we summarize in Section 8. In this work, we explore these techniques and choose the one that achieves the goals specified at the end of Section 2.

Paper Contributions
The primary goal of this work is using machine intelligence to build a hybrid memory page scheduler that can bridge the performance gap between the current state-of-the-art History and the ideal but unrealistic Oracle page scheduler. We build a new page scheduler – **Kleio** – and we answer important questions concerning how to achieve an *effective* solution (i.e., one that maximizes the extent to which the performance gap is bridged), and a *practical* solution (i.e., one that can be realized while expending only a controlled or limited amount of resources on the typically compute-intensive machine intelligence processing tasks).

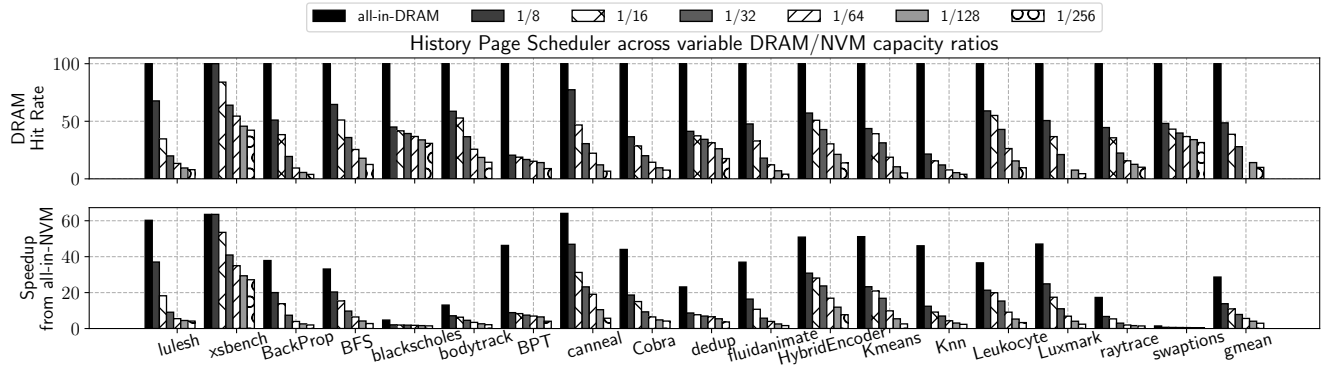
The specific contributions of this paper are the following:

- *Gap in current solutions*: We show the significant room for application performance improvement that is feasible in hybrid memory systems via clever data placement. This is due to the fact that predominantly used solutions, which look at recent memory access activity, are not computationally robust so as to capture complex page access patterns (Section 2).
- *MI-based page scheduling*: We identify Recurrent Neural Networks (RNNs) as an effective and practical technique for the page scheduling problem (Section 3). We show that RNN training on a per application page granularity is highly accurate and leads to significant performance improvements even when applied to a subset of pages (Section 4). While not exhaustively exploring all possible DNN algorithms, we present insights on the important tradeoffs that must be considered when selecting an MI approach: its computational and space complexity and its applicability for the feature set which describes the page scheduling problem.
- *Kleio*: We design **Kleio**[1], a practical, hybrid MI-based page scheduler. Kleio is hybrid because it combines existing history-based page scheduling, when such more lightweight methods are effective, with RNN-based machine intelligence, when history-based methods fail. Kleio is practical because it incorporates a new method for identifying pages where MI-based scheduling leads to most significant performance boost and prioritizing the use of system resources for these pages (Section 5).
- *Performance improvements*: Using a range of workloads from popular suites, we show that Kleio can bridge on average 80% of the performance gap, that exists between the history-based page scheduling and oracular knowledge of the access pattern of a small set of cleverly selected application pages (Sections 6 and 7).

[1]The name is inspired by ancient Greek mythology, where Kleio is the muse of history, daughter of Mnemosyne, goddess of memory.



(a) The Oracle page scheduler periodically migrates application pages such that DRAM hosts the pages with the highest access counts in the *current* scheduling epoch until capacity is full.



(b) The History page scheduler periodically migrates application pages such that DRAM hosts the pages with the highest access counts in the *previous* scheduling epoch until capacity is full.

Figure 2: Application performance for decreasing ratio of DRAM to NVM and fixed overall capacity to be the per application memory footprint. Section 6 includes detailed explanation of the experimental methodology.

2 MOTIVATION

We first provide experimental results to illustrate the scope of the problem addressed with Kleio. The goal is to illustrate the importance of page scheduling for different applications and the gap that exists with current history-based approaches. We use the applications summarized in Table 1, the experimental methodology is described in detail in Section 6, and the scheduler description is introduced in Section 1 as well as summarized in the caption of Figure 2.

Figure 2a shows the performance achieved by an **Oracle page scheduler** across decreasing availability of DRAM capacity. Even in the case of a-priori knowledge of the workload's access pattern, the restricted DRAM capacity can severely impact performance, especially when it is available only in smaller amounts (e.g., 1/256 DRAM/NVM ratio). We also validate the observation [10] that the use of the minimum necessary DRAM capacity that is able to host the hot pages across the scheduling epochs (i.e., 1/8 in our case) can provide almost the same performance as if having infinite DRAM capacity (i.e., all-in-DRAM).

Figure 2b shows how the placement methodology of the current state-of-the-art **History page scheduler** can reduce performance up to **55%** (in the case of `lulesh`) and 13% on average. This is due to the fact that the history-based scheduler is built on the observation that applications preserve their page access pattern for certain time intervals, which may span across multiple scheduling epochs. Although this leads to good page placement decisions during such epochs, it fails to capture changes in the workload's memory access behavior. For example, there are times where the subset of hot pages may be completely disjoint between consecutive scheduling epochs, as the application transitioned into computation that involves data allocated in different memory areas. In this case, the performance impact is significant and makes a case for more intelligent data management using clever extrapolation of the past memory access pattern and not just the immediately observed behavior.

Takeaways

We observed that even though restricted DRAM capacity can potentially reduce application performance, the current state-of-the-art page scheduling methodology is not intelligent enough to capture all the necessary past information needed for predicting future memory access behavior, which will allow for timely data placement in DRAM. To address this, we choose to explore machine intelligence techniques given their ability to learn complex combinations of multi-featured information.

Application	Suite	Domain	pages (4 KB)	Sched. Epochs
Lulesh	CORAL	Hydrodynamics	847,252	206
XSBench	CORAL	Monte Carlo	136,098	856
blackscholes	PARSEC	Finance	8,033	302
bodytrack	PARSEC	Comp. Vision	13,259	389
canneal	PARSEC	Engineering	56,974	398
dedup	PARSEC	Storage	131,259	657
fluidanimate	PARSEC	Animation	54,286	333
raytrace	PARSEC	Visualization	22,890	347
swaptions	PARSEC	Finance	12,633	491
BackProp	Rodinia	Pattern	35,083	117
BFS	Rodinia	Graph	27,396	26
BPT	Rodinia	Filesystems	142,923	485
Kmeans	Rodinia	Data Mining	70,783	87
Knn	Rodinia	Data Classifier	84,691	118
Leukocyte	Rodinia	Medical	56,580	180
Cobra	Windows	Video Transcode	83,720	168
HybridEncoder	Windows	Video Transcode	73,787	178
Luxmark	Windows	Image Creation	53,491	108

Table 1: Workloads used for evaluation. Number of pages × 4 KiloBytes will be the total application memory footprint. Scheduling epochs is the number of times that the page scheduler was periodically invoked within the application runtime, so as to reposition pages across the hybrid memory subsystem.

We aim to achieve two important **goals**:

1. Bridge the performance gap between the Oracle and History page schedulers.
2. Deliver low training and inference times by reducing the input problem space. This would allow the approach to be possibly integrated in an online solution.

In doing so, we contribute answers to the following **questions**:

1. Which machine intelligence technique to use (Section 3)?
2. How should we formalize the data input to the machine intelligence algorithm, so that it adheres to the purpose of predicting page access behavior to be used by a page scheduler (Section 4)?
3. How can we reduce the input problem space? How many are the pages whose timely placement in DRAM significantly boosts performance, while the History scheduler fails to properly manage them? Do all pages actually need machine intelligence based management (Section 5)?

3 MACHINE INTELLIGENCE BACKGROUND

In this section, we explore the machine intelligence techniques that seem to be a good fit when designing an application page scheduler for data management over hybrid memory systems. Our goal is to design a page scheduler that can learn more cleverly through past information and make more intelligent page placement decisions across the scheduling epochs, compared to the existing History page scheduler, as depicted in Figure 1 and described in Sections 1, 2.

3.1 Reinforcement Learning

First, we explored deep reinforcement learning [13, 22, 23], a machine intelligence technique that enables an agent to learn through taking actions in a defined environment, in order to maximize a reward entity via the received feedback.

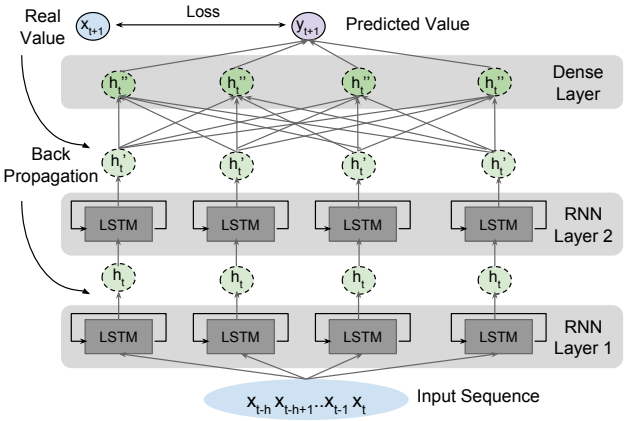


Figure 3: Example layout of a Recurrent Neural Network (RNN), using Long Short Term Memory (LSTM) neurons.

In more detail, the page scheduler (*agent*) periodically interrupts the execution of the application to take an *action*, that is to migrate pages across the memory components. Then, the application resumes execution (*environment*) and during the next scheduling epoch (interrupt) the page scheduler receives its *reward*, that is the DRAM hit rate with the most recent page placement (*state*). In this way, the page scheduler learns the dynamic data layout that optimizes application performance across its runtime.

Why it is not a good fit. Although the approach of reinforcement learning seems to be a great fit into the problem description of a hybrid memory page scheduler, it cannot be realistically adapted. This is due to the prohibitively large amount of possible actions the agent (page scheduler) can take. More specifically, a single action of a page scheduler involves taking a placement decision for each individual application page. For example, if there are two memory components and N pages, then there are 2^N possible placements, thus actions to choose from. Considering Table 1, that summarizes the number of pages across our pool of applications, N can be in the order of hundred thousands. In conclusion, the problem space becomes not only exponential, but also depends on the number of application pages, which made us drop the approach of reinforcement learning for the context of our problem.

3.2 Recurrent Neural Networks

Another machine intelligence approach, which seemed appropriate for the purpose of the hybrid memory page scheduler, is Recurrent Neural Networks (RNNs). Different from reinforcement learning, where interaction with an environment facilitates learning, RNNs are able to find long-term dependencies in a sequence of data points and make predictions about future data behavior.

In the context of the page scheduler, these data points can be the sequence of pages accessed throughout an application execution time interval. The page scheduler can deploy an RNN in order to learn the page access pattern and make predictions about future page accesses. Using those predictions the page scheduler can determine which pages should be prioritized for allocation in the

most appropriate memory component. For example, future highly accessed pages should be allocated in the lowest access latency memory technology. We choose to adapt this machine intelligence technique, since it has already been used to solve similar problems, like hardware memory prefetching [12]. In contrast with reinforcement learning, where the problem space was growing exponentially to the number of application pages, in the case of RNNs it grows linearly with the number of pages. Furthermore, in Section 4 we show how it can be significantly reduced for the purpose of fast and efficient learning.

RNN Functionality. Next, we present the internal functionality of RNNs on a very high level. Currently, a widely used type of RNN is the Long Short Term Memory (LSTM) Network, that given a sequence of data points from time $t - h$ up to time t, can make a value prediction for time $t + 1$, where h is the length of retained history. For example, if the sequence represents the weather forecast of a city from April to November, the LSTM can make a weather prediction for December. In more detail, a single LSTM neuron takes the input sequence and converts it into an internal state h_t, via a non-linear combination of the weights and biases of its internal 'gates'. There are the 'input', 'output' and 'forget' gates that dictate what information gets filtered from the input and propagated towards the output. In this way, a single LSTM neuron is able to capture past data information into an internal state representation and make predictions about future data points.

An RNN can be constructed via the combination of multiple LSTM neurons on a single layer, stacked LSTM layers together with regular Dense layers, as depicted in Figure 3. The input sequence is split into subsequences of *history length* h, in a rolling window fashion. During a *training epoch*, all input subsequences are fed into the network, which then makes a single value prediction for each subsequence. The difference between the predicted and actual values is captured through the *loss function* and *back-propagated* into the network, where its weights and biases are getting updated according to the *learning rate*. Training can terminate when there is no reduction in the loss, thus the network cannot make any predictions closer to the actual value. In Section 6 we describe the network layout, hyper-parameter values and further fine-tuning techniques that will facilitate learning for the provided input data.

4 NEURAL NETWORK INPUT

When using neural networks, an important step is choosing the features which describe the problem and are to be used as inputs. In this section, we discuss the representation of the data sequence related to memory access behaviors to be fed into the RNN and the interpretation of the predicted value, as this is crucial for the training time and accuracy of the generated model. We further explore possible ways to reduce the input problem space and enable faster and more resource-efficient learning.

Input Data. The data we have available for each application is a memory access trace, as depicted in Figure 4. More specifically, it is the sequence of the page accesses that were serviced from main memory and not the processor's hardware caches, as they happened throughout the application run time. In Section 6 we describe in detail the way we acquire the trace and the exact information it contains.

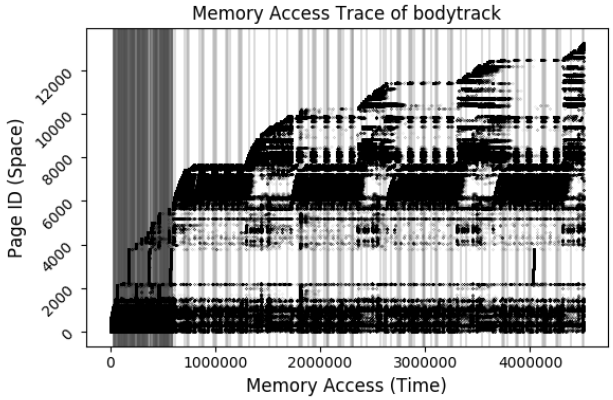


Figure 4: Example memory access trace, from the PARSEC suite. For every consecutive memory access (x-axis) we plot the page accessed (y-axis). The vertical gray lines correspond to the scheduling epoch time intervals.

Learning Objective. The aim of the RNN training is to be able to make predictions with respect to the number of future memory accesses, so as to aggregate the accesses on an application page granularity and then determine an ordering of heavily accessed pages. These predictions need to happen periodically, when the page scheduler is invoked, so that the appropriate page migrations are determined and executed. That is future hot pages need to be migrated to the memory technology component with lowest access latency.

Training Time. One of our main considerations is to enable fast learning via reduced training times and resource utilization optimized techniques. The duration of training models can be critical when considering use of machine intelligence in systems solutions, which to be practical, must operate within limited time and computational resource budgets. Undoubtedly the use of computationally robust technologies, like GPUs, TPUs, custom RNN accelerators, can accelerate learning. However, in this paper our primary goal is to explore ways to enable faster learning via the training methodology, that can further be boosted via appropriate hardware.

4.1 Across Pages Prediction

The most intuitive way to learn from a memory access trace is to feed it 'as-is' into the RNN, following the x-axis in Figure 4. In this case, the RNN looks into a subsequence of page accesses and predicts the page to be accessed next. Such an RNN use case is used by Hashemi *et al.* [12], for the purpose of prefetching future memory address accesses.

This approach has several *limitations:*
1. Large training time. To begin with, the input trace usually contains millions of memory accesses, especially at the data input scales of High Performance Computing applications. This makes training time prohibitively large, in the order of couple days, at least when using the hardware setup described in Section 6.

2. Low prediction accuracy. Furthermore, when the output value space is significantly large (number of different pages), the RNN prediction accuracy tends to be low. Neural networks work better with normalized inputs (e.g., between 0 and 1 [12]). However, when normalizing hundred thousand values in such a way (total number of pages according to Table 1), there will be vast information loss. This is the reason why Hashemi *et al.* [12], choose to reduce the output value space (number of different memory addresses), by discretizing it into frequently appearing values (classes), and training different RNNs across clusters of the address space covered by the application. Most importantly, they accept top-k predictions at a time, so as to increase the chances of a correct prediction. Although this is acceptable for the purpose of prefetching, it is not the case for a page placement decision, where a single prediction is needed, in order to accumulate the number of per page accesses.

3. Not an exact fit for the page scheduler description. As described in Section 1, the page scheduler operates periodically, aggregating the per page access counts during an application runtime interval referred to as scheduling epoch. Then the scheduler will determine the appropriate page ordering and issue the necessary migrations across the memory components. However, the number of memory accesses differs across the scheduling epochs, as it is visible by the vertical lines in Figure 4, where only 10% of the total memory accesses happened during the first half of the scheduling epochs. This is subject to the code executed during that time with respect to its computation to data access ratio and the technology parameters of the processor and memory regarding the time it takes to execute an operation, load data, etc. Throughout our application pool, we observe that just 10% of the total memory accesses happen, on average, throughout the first 37% of the scheduling epochs. Thus, there is no way to know before-hand how many accesses are going to happen in the next scheduling epoch, that is how far in the future the RNN should make predictions for (unless we train a different RNN for that purpose!).

In conclusion, we reject the idea to treat the input access trace as-is, given the restrictions described above. Next, we will see how we can extract the necessary information from the trace, so as to enable faster and accurate learning, that is also more suitable for the functionality of a page scheduler.

4.2 Per Page Prediction

Instead of predicting *which* page is going to be accessed next (across pages prediction), we flip the problem and explore the case of predicting *when* a page is going to be accessed next (per page prediction). So instead of predicting the y-value following the x-axis, we take each y-value (page) and predict the sum of accesses across the scheduling epoch intervals on the x-axis. Thus, we propose training individual RNNs for every single application page. So, we feed into the per page RNN the sequence of access counts across the scheduling epochs and predict the number of accesses that the page will receive in the next scheduling epoch. In contrast with the prediction across page, the per page prediction:

1. Fits the page scheduler description. The above transformation of the input access trace fits exactly the functionality of the page scheduler, which will aggregate the page access counts on a scheduling epoch interval, so as to order frequently accessed pages and appropriately migrate them across the hybrid memory components.

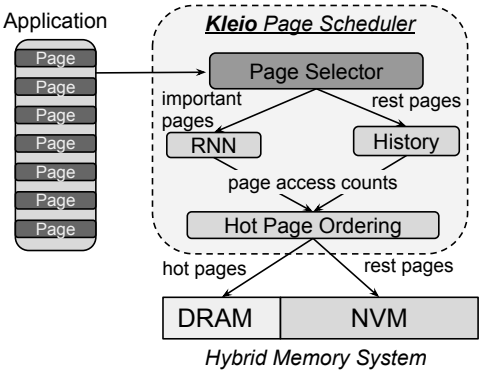


Figure 5: Kleio is a hybrid memory page scheduler, that combines the current state-of-the-art page placement methodology together with machine intelligence based management of the page subset, whose timely placement in the appropriate memory component is crucial for application performance.

2. Enables high prediction accuracy. Depending on the epoch duration and hotness of the page, the maximum number of accesses per epoch is in the order of hundreds, which is orders of magnitude less than problem space that the prediction across pages needed to capture, normalize and predict. Thus, this output value range is more suitable for RNN training.

3. Allows for low training times. Having a different RNN model per page, when the total number of pages can be in the order of hundred thousands, is similar to having a single RNN model that makes predictions across all these pages, as described earlier, since the input problem size remains the same, as depicted in Figure 4. Similarly to clustering techniques of the address space into memory regions and focusing on the frequently appearing memory addresses, as Hashemi *et al.* [12] did, there is scope to focus on the pages that are critical to application performance, which will significantly reduce the number of RNN models and overall training time, thus resource consumption.

In Section 5 we describe in detail, the methodology of selecting the most appropriate pages for training with respect to reaching a desired level of application performance.

5 SOLUTION

We propose **Kleio**, a page scheduler for hybrid memory systems, that leverages the existing state-of-the-art data management solutions and optimizes application performance by delivering machine intelligence based placement decisions for a cleverly selected page subset.

Kleio Overview. Figure 5, summarizes Kleio's internal functionality. Kleio takes the following actions periodically, that is on every scheduling epoch:

1. Identifies the subset of application pages that are important to performance, through its page selector component, described in detail later on.

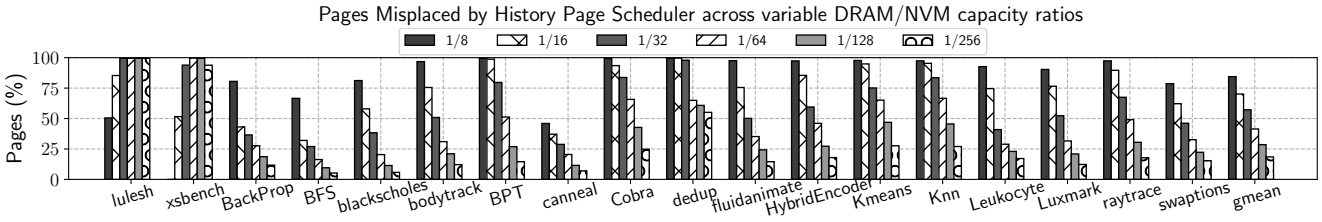


Figure 6: Percentage of pages misplaced at least one time across the scheduling epochs by the History page scheduler. This is the set of pages that need machine intelligence based management. This observation is crucial since it highlights that the problem space of per page RNN training can be significantly reduced as the size of available DRAM does.

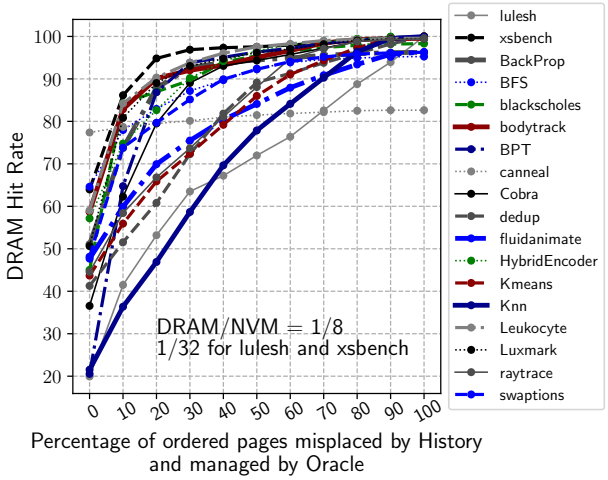


Figure 7: DRAM hit rate when an Oracle Page scheduler manages the misplaced-by-History pages and the History page scheduler manages the rest. Pages are ordered in descending performance benefit. Clever management of even a small percentage of these pages, can give most of the performance benefits we would have by managing cleverly all pages.

2. Trains an individual RNN for each of the important pages, in order to predict the page access counts for the following scheduling epoch.
3. For the rest of the pages, Kleio assumes that they will preserve their access counts in the following epoch, as the current state-of-the-art History page scheduler does, as described in Section 1.
4. At this point, Kleio has accumulated the per page access counts for the next scheduling epoch. It then orders the pages in descending access frequency order, prioritizing DRAM allocation for the hot pages, until capacity is full. This methodology is predominantly used for performance optimal data tiering in hybrid memory systems [9, 11, 27].

Following the above steps, Kleio is able to bridge the performance gap between the Oracle and History page schedulers, as described in Section 1. Full evaluation of Kleio, with respect to the machine intelligence accuracy and application performance optimization is done is Section 7.

5.1 Page Selector

We first describe the page selector component in Kleio. Its design is driven by the following observations regarding the importance of correct page placement to application performance:

- There is only a certain subset of pages that needs more clever data management, than what the existing history-based solutions can provide. That subset is significantly small for limited DRAM capacity.
- Pages that need machine intelligence based management, can be ordered with respect to the performance impact of their placement into the appropriate memory component. We define a benefit metric that enables the page ordering, prioritizing pages with high access counts and number of misplacements by the History page scheduler.
- Intelligent management of the pages following the aforementioned ordering does not correspond to linear performance improvement. In contrast, intelligent placement for only (a small) part of them can bring most of the performance benefits we would get by applying intelligent placement across all application pages.

We define a 'misplacement' of a page by the History scheduler, when at the start of a scheduling epoch, a page was supposed to be allocated in DRAM, but it was not, because of wrong hotness prediction. Figure 6 depicts the percentage of application pages, which are misplaced by the History page scheduler, at least during one scheduling epoch, across reducing DRAM capacity. This signifies the set of pages that need more clever management. In combination with the actual per application page count summarized in Table 1 and the limited DRAM capacity, the number of such pages can be in the order of hundreds. This drastically reduces the problem space of RNN training.

However, even by reducing the number of such pages, there still may not be enough resources or time to train per page RNN models. Thus, there needs to be a priority ordering of these pages, so as to cleverly manage those that can give the biggest application performance boost, when timely placed into DRAM. For this reason, we capture the importance of a page in the *benefit* that its correct placement would provide to application performance. The benefit increases with the hotness of a page, similarly to prioritizing frequently accessed pages for DRAM allocations across the scheduling epochs. However, we also need to take into account the number of misplacements-by-the-history-scheduler each page received, as the timely placement of a page in DRAM together with its hotness, will

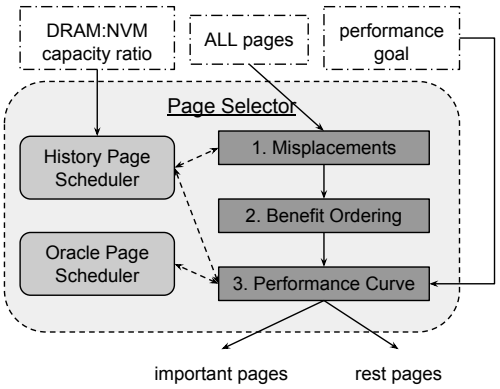


Figure 8: Kleio's page selector component is able to identify the pages whose machine intelligence based management will bring the highest application performance improvements, while enabling focused and practical RNN training.

boost performance. To this extent, we define the following benefit factor for prioritizing the pages in need of RNN training.

`Benefit = Number of accesses` $\times$ `Number of misplacements`

Next, we capture the range of application performance boost we would get, if we could manage part of the aforementioned misplaced pages with the Oracle page scheduler and the rest with the History page scheduler, since it already places hot pages in DRAM in time. This will set the upper limit of the performance boost we can get with RNN training of the misplaced pages. Figure 7 captures this performance improvement. When the Oracle scheduler manages 0% of the misplaced pages, it is equivalent to all pages being managed by the History scheduler, thus is the lowest bound of performance. In contrast, when the Oracle scheduler manages 100% of the misplaced pages, it is equivalent to the Oracle managing all the application pages, since the rest of the pages were not misplaced by the History scheduler. That sets the upper bound of performance we can have on a per application basis. We observe a non-linear relation between the set of pages and the performance enhancement. This is due to the page ordering with respect to the defined benefit factor, that is able to prioritize hot pages, whose timely DRAM allocation guarantees significant performance improvement. For example, in the case where the curve shows a distinct knee, the pages after the knee, received far less accesses, thus their timely DRAM placement will bring trivial benefit to the DRAM hit rate.

Kleio's page selector component captures the above observations and provides insight into the relation between the number of pages that need RNN training together with a best case scenario of corresponding application performance improvement. Figure 8 summarizes the work flow of the page selector component.

6 METHODOLOGY

6.1 Applications

Table 1 summarizes the set of workloads we used to motivate and evaluate Kleio, spanning across domains with representative computation kernels and stressing different components of the system (e.g., memory, CPU, GPU). We included workloads from the CORAL [1] suite, the PARSEC [3] suite utilizing the simlarge input sizes, and Rodinia [5], with the default input data sizes. Finally, we also included few Windows desktop applications.

Concerning the memory footprint of these applications, Table 1 includes this information as a multiple of 4 KB pages, that gives a range of couple hundred MBs. Such significant memory footprint (e.g., relative to the cycle-level memory simulations) is necessary for our analysis, so as to capture the use case where the data will span across multiple main memory components, due to the limited capacity of available DRAM in future hybrid memory systems.

Regarding the application runtime, it is again summarized in Table 1, as a multiple of the scheduling epoch intervals, when the page scheduler is periodically triggered throughout application execution. Our applications serve a variety of short and long running executions. Due to the difference in the trace collection methodology, for the CORAL workloads the scheduling epoch interval is 1 second, whereas for the rest is 0.01 seconds.

6.2 Memory Access Trace Collection

For each application we collect detailed traces of the data accesses that missed the last level of processor hardware caches and resulted in main memory accesses. For the CORAL workloads we used the Instruction Based Sampling (IBS) that is available on AMD's processors. This mechanism samples every Nth micro-operation, that goes through the processor's pipeline, out of which we filter the loads and stores. For the rest of the workloads, we collected unsampled traces for memory accesses that miss the last level cache on a system with an AMD A10-5800K APU clocked at 3.8GHz and 16GB memory. The information included for each individual access is a timestamp, the physical and virtual memory address, the CPU core ID, the application thread ID, whether the access was a load or a store and a hit or miss. For the purpose of our analysis, we extract the 4 KB virtual page ID, that corresponds to the virtual memory address accessed and we group memory accesses into scheduling epoch intervals according to the timestamp, as depicted in Figure 4.

6.3 Hybrid Memory System Simulation

We simulate a hybrid memory system that contains a fast memory component (i.e., DRAM) and one with lower access latency (i.e., NVM). Both memory technologies serve as flat main memory, as they are part of a continuous physical memory address space. Table 2 summarizes the technology parameters of the simulated memory types. The capacity of the memory system is assumed to be the application's memory footprint. For example, when we refer to a DRAM/NVM capacity ratio of 1/16, we mean that DRAM will have space to accommodate 1/16 of the application pages and NVM will service the rest.

Apart from gathering the DRAM hit rate as an application performance metric, we also use the analytical model used by Meswani *et al.* [20] to extrapolate the application runtime, based on the number of accesses that are serviced from DRAM and NVM appropriately. In the case of the CORAL workloads, the number of accesses is properly adjusted based on the sampling rate. The model uses the Leading Loads method, which splits the application runtime into the time to perform computations and the time to satisfy memory requests, via the use of hardware performance counters. Regarding the time to service a memory request, the method maps it to the

Technology	R/W BW (GB/s)	Seq. & Rand. R/W Latency (ns)
DRAM	19.2/19.2	8/8 & 50/50
NVM	10.24/1.024	8/8 & 100/1000

Table 2: Technology parameters used in the simulated hybrid memory system, differentiating for Reads (R) and Writes (W) and sequential versus random accesses.

time spent servicing the leading (first out of many) load request that misses the last level hardware cache. This load time depends on the memory technology that serviced the request (e.g., DRAM versus NVM), whose differences are summarized in Table 2. This gives us a worst case performance estimate, since it does not take into account actions that reduce latency, such as parallel computation or prefetching. Also, we assume dedicated DMA engines that allow seamless page migration, which is overlapped with the computation, as explored in [14, 19].

6.4 Neural Network Details

Neural Network Layout. Figure 3 gives a visual representation of the RNN we deployed, consisting of LSTM neurons. The network consists of two stacked RNN layers with *128* LSTM neurons each, followed by a Dense Layer. The history length is *16*, thus the input data series is split in sequences of length 16, on a rolling window fashion, while *70%* of them are used as a training dataset and *30%* of them for validation. The neural network tries to minimize the *mean squared error* (loss) between the predicted and actual values, using the Adam [16] optimizer on a learning rate of *0.001*. The model training stops, if the loss for the validation dataset is not reduced for *20* consecutive training epochs. The duration and accuracy of the trained models is reported in Section 7.

Data Manipulation. As described in Section 4.2, the RNN input corresponds to a sequence of per page access counts during consecutive scheduling epochs, while the output is the predicted number of accesses the page will receive during the next epoch. The predicted number will then be used by the page scheduler to determine the hotness order across all pages. Thus, there is room for the prediction to be slightly different than the actual number of accesses, as long as it will not influence the hotness order of the page, and therefore its placement decision, on the particular scheduling epoch.

Therefore, we normalize the input sequences between 0 and 1, since RNNs work better in this case as observed by Hashemi *et al.* [12] and then denormalize the data for the final prediction. Different from [12], there is no need for us to make predictions over distinct integers, treating the prediction problem as classification. Our experiments with the classification approach, highlighted the possibility of misprediction with a great margin from the actual value and gave reasoning as to why Hashemi *et al.* [12] chose to consider top-k predictions at a time. Although this approach works great with the prefetching logic, where more data can be prefetched even if they do not end up being accessed, this is not necessary for the purpose of our predictions.

It is important to observe that, even though the input data (memory access trace) is the same between this work and [12], the prediction use case transforms the way they should be manipulated for RNN training and the accepted level of prediction accuracy.

Implementation. We use the Keras [6] high level API to deploy the described RNN layout, using the existing implementations for the LSTM neurons, the network layers connectivity, the Adam optimizer and model training, applying any default hyper-parameter values if not explicitly mentioned above. The backend RNN execution engine is Tensorflow [2].

Hardware Testbed. We conduct experiments using an AMD machine with 512 GB memory and 64 Opteron™ 6370P CPU cores of 2 GHz each. CPUs have been used to accelerate RNN-based deep learning models [32]. Kleio speeds up the training by intelligently selecting to train the application pages that will bring actual performance benefits. Instead, a more naive approach would rely on accelerators and rack-scale size machines in order to accommodate RNNs for all pages, wasting resources for training models whose predictions have trivial performance impact or can be achieved by simple history-based policies.

7 EVALUATION

In Section 5.1 and in particular in Figure 7, we showed the trend of performance improvements Kleio can provide, assuming oracular knowledge of the access counts of the pages that are in need of machine intelligence based placement. In this section, we evaluate Kleio with respect to the actual application performance improvements it can provide. We report how close to the Oracle page scheduler Kleio can perform, when managing the pages that are misplaced by the History page scheduler. We also summarize the accuracy of the RNN predictions and the RNN training overheads. Together with the achieved performance, these make a case for Kleio's practicality.

7.1 Application Performance

First, we evaluate the accuracy of Kleio's RNN training with respect to the corresponding application performance improvements, which is what Kleio promises to deliver. As a reminder, Kleio identifies the pages that are misplaced by the History page scheduler and applies RNN training in order to get predictions of their per epoch access counts and determine the global page hotness order for prioritizing DRAM allocations. If the RNN predictions are extremely accurate, then it would be equivalent to having an Oracle page scheduler manage the misplaced pages. To this extent, Figure 9a depicts the performance that Kleio can achieve when applying RNN training to **100 pages** in the order defined by its page selector component, for a given DRAM/NVM capacity ratio. We fix DRAM/NVM=1/32 for the CORAL workloads and DRAM/NVM=1/8 for the rest, which is the capacity ratio for which the clever management of even a small number of pages, can bring significant performance improvements (Figure 7). Performance is normalized between 0%, when all pages are managed by History page scheduler and 100%, when the selected pages are managed by Oracle and the rest by History. In this way, we can understand the degree to which the RNN predictions are sufficiently accurate, so as to provide all the possible performance improvement.

We observe that in most cases, the RNN predictions are sufficiently accurate to bring **80%** of the possible performance improvement, on average and more than **95%** for half of the applications that we considered. Unfortunately, there are cases such

as bodytrack and raytrace, where less than 50% of the possible speedup is achieved, in which case more pages need to be trained so as to further provide significant speedup.

Overall, we prove that the accuracy of the RNN predictions is such that it can deliver application performance similar to what would be possible with oracular knowledge of the access frequency. Kleio's page selector is useful, so as to determine the number of pages that is necessary to train in order to observe significant performance improvements.

7.2 Prediction Accuracy

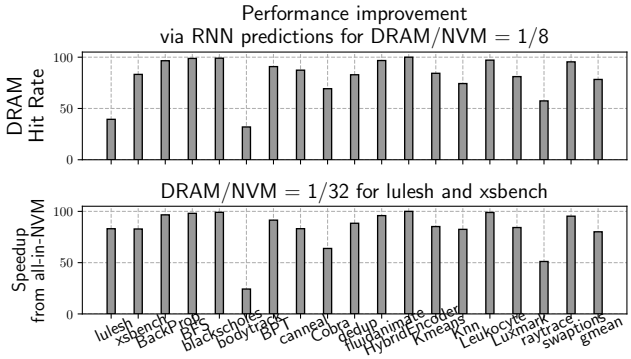
We next present the actual prediction accuracy of the per page RNN training. Figure 9b depicts the distribution of the Mean Absolute Error (MAE), in boxplot representation, between the cumulative per epoch page access counts and the actual values, across the trained application pages. For example, mean MAE of 30, means that the RNN predicted 30 more accesses on average per epoch per page. On the same graph, we treat the decisions of the History page scheduler also as predictions and plot the corresponding MAE. The History page scheduler predicts that on the next scheduling epoch a page will receive the same access counts as to those of the current epoch.

As expected, the History prediction can be far from reality, as it is common for a page to convert from being frequently accessed to not being accessed at all on two consecutive epochs, thus the prediction MAE can be significantly high. In contrast, the RNN is able to make better predictions via the efficient LSTM learning, although still they may seem not as accurate enough. However, as explained in Section 6, even if the per epoch access count prediction is not extremely accurate, as long as it does not affect the correct global page hotness order and actual page placement, there will be no application performance impact of the prediction. This is highlighted in Figure 9a, where for example Luxmark has a mean MAE of 50, though still achieves 85% of the possible performance improvements.

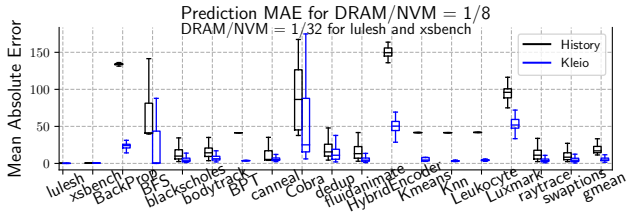
Figure 9c, further strengthens the above statement by showing the percentage reduction of page misplacements achieved by Kleio for the selected trained pages, compared to the History page scheduler across all pages. Although, Kleio still misplaces the selected pages on some scheduling epochs, the per page access count during those epochs is not big enough to drastically impact the DRAM hit rate. Thus, Kleio manages to reduce on average 85% of the selected pages misplacements across the application lifetime.

7.3 Resource Utilization

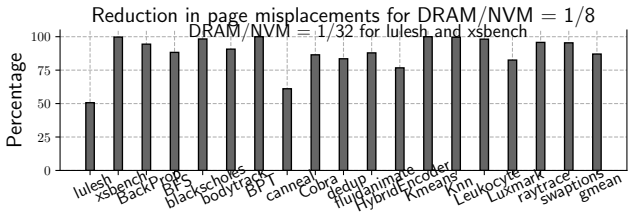
As summarized in Section 6, RNN training goes on until there is no further reduction of the loss over the validation data for a certain number of training epochs. The duration of the training is primarily affected by the network layout itself, that is the hyperparameter values and the length together with the number of the input sequences. Thus, the more training data the longer it takes to learn. Since we perform training on a per application and per page granularity, the number of input sequences is the number of scheduling epochs, divided by the history length hyperparameter. Looking back at Table 1, this number will be in the order of couple hundreds, which enables fast training times.



(a) Performance achieved by machine intelligence based management of the first 100 most important to performance pages, while a History page scheduler manages the rest. Hit rate is normalized between 0, that is the worst-case where all application pages are managed by History and 100, that is the ideal case where Oracle manages these 100 pages. Kleio can deliver over 80% (gmean) of the performance improvements that an Oracle page scheduler would for the selected pages.



(b) Prediction accuracy of the number of access counts across the scheduling epochs for the selected trained pages.



(c) Reduction in the number of page misplacements via the achieved RNN prediction accuracy, compared to the History page scheduler.

Figure 9: Evaluation of the application performance Kleio can deliver.

More specifically, we report the following average metrics across pages and across applications, for the given hardware testbed described in Section 6. Training lasts on average for **120 training epochs**, that translates into a time duration of **2 hours** per model, when all models are trained at the same time, utilizing all system's resources. As far as memory utilization during training is concerned, the maximum observed per model was in the order of **tens of GBs**. Finally, regarding the storage overheads of saving the models after training, for the purpose of future inference and analysis, using the Hierarchical Data Format (.hdf5) available from the Keras library, it was less than **0.5 MB** per model. Regarding the resource utilization for the purpose of inference, it was trivial and the duration instantaneous (3-4 seconds).

Putting all this information together, there is no doubt that the hardware resource requirements of RNN training are significant, especially as far as memory consumption is concerned. However, training times in the order of couple hours are generally considered to be low, for machine intelligence purposes. Furthermore, the training time can be further reduced using more computationally robust hardware. Either way, the user may be limited with respect to how many per page models can train, given the available system resources.

Kleio has provisioned for the case of limited hardware resources through its page selector component, that provides the user with information regarding which pages to prioritize for RNN training and the corresponding expected application performance improvements.

Reaching our initial Goals.

1. Kleio promises to bridge the performance gap between the Oracle and History page schedulers, delivering on average 80% of the theoretically possible performance when managing selected pages, through the achieved RNN prediction accuracy.
2. Kleio delivers low training and inference times, via deploying RNN models for cleverly selected application pages, whose timely placement in DRAM significantly boosts performance. Kleio shows that not all pages are in need of intelligent data management, drastically reducing the input problem space.

8 RELATED WORK

Kleio is a research artifact that utilizes neural networks in order to enable learning of a workload's memory access behavior for the purpose of application page placement across a hybrid memory system. In this section, we describe some of the machine intelligence approaches used in the system's community, focusing either on other relevant problems or just other aspects of data management in hybrid memory systems.

Regarding the usage of RNNs in the system software stack or in hardware, there has already been a significant amount of research. To begin with, RNNs have been proposed for the purpose of memory prefetching by Hashemi *et al.* [12] as well as Zeng *et al.* [31]. We have made multiple points in Sections 4 and 6 about how differently we deploy RNNs and the importance of considering the manipulation of the input data to be appropriate for the use case of the trained model. Concerning other use cases of RNNs, the authors of Desh [8] deploy them in order to predict node failures in Supercomputing environments, so as to timely migrate computation towards live nodes. In addition, RNNs can be utilized in order to learn I/O block level access patterns, so as to optimize the performance of flash storage device usage [4]. Furthermore, RNNs could also be used over standard resource usage statistics and kernel-level events, so as to predict future resource usage of applications [26]. Finally, the authors of DeepCache [24] build a content caching framework utilizing RNNs and in particular the LSTM Encoder - Decoder model.

Regarding hybrid memory data management, we already refer to a significant number of solutions without machine intelligence in Section 1, such as [20, 27, 28]. As far as proposals with machine intelligence are concerned, the authors of Tahoe [29] explore supervised machine learning techniques (multiple linear regression and artificial neural networks), in order to predict application performance baselines that will be part of the data object placement cost across the hybrid memory components. Moreover, an alternative approach to hybrid memory and distributed memory designs is to leverage the knowledge about specific application algorithms to direct data placement, rather than make the scheduling decisions based on memory access trace data (thus treating the workloads as a black box). Additionally, Wu *et al.* [30] demonstrate that algorithm features, common numerical operations, and algorithm structures can be leveraged to direct data placement for conjugate gradient, fast Fourier transform, and LU decomposition for a matrix. They also introduce a hardware customized DMA mechanism for bulk data movement which is complimentary to this work. The k-means NUMA Optimized Routine (knor) library [21] optimizes k-means for modern NUMA architectures and minimizes synchronization barriers.

With respect to similar system problems, Selecta [17] utilizes latent factor collaborative filtering, in order to find the configuration of cloud compute and storage resources that provides optimal cost-to-performance trade-offs. Finally, Kraska *et al.* [18] demonstrate the benefits of having machine intelligence based data indexing and argue that the replacement of parts of the data management stack with machine Intelligence based components will provide significant performance benefits.

9 SUMMARY

We present Kleio, a page scheduler with machine intelligence for applications that execute over hybrid memory systems. Kleio leverages the current state-of-the-art scheduling methodology based on the intuitive observations that frequently accessed pages need to be placed in the fastest memory component and the fact that such pages will remain frequently accessed for a period of time. Going a step further than existing solutions, Kleio applies recurrent neural network training to detect page access behavior, that cannot be captured by the above observations, such as sudden changes in the access frequency of a page. Furthermore, Kleio drastically reduces the number of pages that need neural network training, by detecting the ones whose clever placement will actually benefit application performance. In this way, Kleio delivers a practical machine intelligence solution and achieves performance improvements close to the ones established by having a-priori knowledge of the workload's memory access pattern.

ACKNOWLEDGMENTS

This work was partially supported by NSF award SPX-1822972, the DOE ECP project on Simple Interfaces for Complex Memories (SICM) and the DOE SSIO Unity project.

REFERENCES

[1] 2018. CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[4] Chandranil Chakraborttii, Vikas Sinha, and Heiner Litz. 2018. SSD QoS Improvements Through Machine Learning. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 511–511. https://doi.org/10.1145/3267809.3275453

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] François Chollet et al. 2015. Keras. https://keras.io.

[7] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 268–280. https://doi.org/10.1145/3132402.3132404

[8] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 40–51. https://doi.org/10.1145/3208040.3208051

[9] Thaleia Dimitra Doudali and Ada Gavrilovska. 2017. CoMerge: Toward Efficient Data Placement in Shared Heterogeneous Memory Systems. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 251–261. https://doi.org/10.1145/3132402.3132418

[10] Thaleia Dimitra Doudali and Ada Gavrilovska. 2018. Mnemo: Boosting Memory Cost Efficiency in Hybrid Memory Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 523–523. https://doi.org/10.1145/3267809.3275465

[11] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[12] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, StockholmsmÃďssan, Stockholm Sweden, 1919–1928. http://proceedings.mlr.press/v80/hashemi18a.html

[13] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *SIGARCH Comput. Archit. News* 36, 3 (June 2008), 39–50. https://doi.org/10.1145/1394608.1382172

[14] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: Smart Allocation and Replication of Memory For Parallel Programs. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 263–276. https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle

[15] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS - OS Design for Heterogeneous Memory Management in Datacenter. In *44th International Symposium on Computer Architecture (ISCA'17)*. Toronto, ON.

[16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 759–773. http://dl.acm.org/citation.cfm?id=3277355.3277429

[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017). arXiv:1712.01208 http://arxiv.org/abs/1712.01208

[19] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 369–383. https://doi.org/10.1145/2980024.2872401

[20] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Vol. 00. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[21] Disa Mhembere, Da Zheng, Carey E. Priebe, Joshua T. Vogelstein, and Randal Burns. 2017. Knor: A NUMA-Optimized In-Memory, Distributed and Semi-External-Memory K-means Library. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/3078597.3078607

[22] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. https://arxiv.org/abs/1706.04972

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[24] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. DeepCache: A Deep Learning Based Framework For Content Caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML (NetAI'18)*. ACM, New York, NY, USA, 48–53. https://doi.org/10.1145/3229543.3229555

[25] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 188–200. https://doi.org/10.1109/PACT.2015.30

[26] Florian Schmidt, Mathias Niepert, and Felipe Huici. 2018. Representation Learning for Resource Usage Prediction. *CoRR* abs/1802.00673 (2018). arXiv:1802.00673 http://arxiv.org/abs/1802.00673

[27] Du Shen, Xu Liu, and Felix Xiaozhu Lin. 2016. Characterizing Emerging Heterogeneous Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/2926697.2926702

[28] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[29] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-volatile Memory-based Heterogeneous Memory for Task-parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 31, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291698

[30] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S. Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2907294.2907321

[31] Yuan Zeng and Xiaochen Guo. 2017. Long Short Term Memory Based Hardware Prefetcher: A Case Study. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 305–311. https://doi.org/10.1145/3132402.3132405

[32] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 951–965. https://www.usenix.org/conference/atc18/presentation/zhang-minjia