Rapid Execution Time Estimation for Heterogeneous Memory Systems Through *Differential Tracing*

Nicolas Denoyelle[1], Swann Perarnau[1], Kamil Iskra[1], and Balazs Gerofi[2(✉)]

[1] Argonne National Laboratory, Lemont, USA
{ndenoyelle,swann,iskra}@anl.gov
[2] RIKEN Center for Computational Science, Kobe, Japan
bgerofi@riken.jp

Abstract. As the complexity of compute nodes in high-performance computing (HPC) keeps increasing, systems equipped with heterogeneous memory devices are becoming paramount. Efficiently utilizing heterogeneous memory-based systems, however, poses significant challenges to application developers. System-software-level transparent solutions utilizing artificial intelligence and machine learning approaches, in particular nonsupervised learning-based methods such as reinforcement learning, may come to the rescue. However, such methods require rapid estimation of execution runtime as a function of the data layout across memory devices for exploring different data placement strategies, rendering architecture-level simulators impractical for this purpose.

In this paper we propose a differential tracing-based approach using memory access traces obtained by high-frequency sampling-based methods (e.g., Intel's PEBS) on real hardware using of different memory devices. We develop a runtime estimator based on such traces that provides an execution time estimate orders of magnitude faster than full-system simulators. On a number of HPC miniapplications we show that the estimator predicts runtime with an average error of 4.4% compared to measurements on real hardware.

Keywords: Memory management · Heterogeneous memory · Machine learning

1 Introduction

As dynamic random-access memory (DRAM) approaches its limits in terms of density, power, and cost, a wide range of alternative memory technologies are on the horizon, with some of them already in relatively large-scale deployment: 3D NAND flash [32], non-volatile memories such as 3D-XPoint [16], spin-transfer torque magnetic RAM [45], and phase-change memory [27]. Moreover, high-performance volatile memories, such as Hybrid Memory Cube [19], high-bandwidth memory (HBM) [21], and Graphics Double Data Rate 6 [22], are

A.-L. Varbanescu et al. (Eds.): ISC High Performance 2022, LNCS 13289, pp. 256–274, 2022.
doi.org/10.1007/978-3-031-07312-0_13

actively being developed and deployed. Resource disaggregation [4], an emerging compute paradigm that has been receiving a lot of attention recently, will further expand the heterogeneous memory landscape.

While these technologies provide opportunities for improving system utilization and efficiency through better matching of specific hardware characteristics with application behavior, at the same time they pose immense challenges to software developers. Management of such heterogeneous memory types is a major challenge for application developers, not only in placing data structures into the most suitable memory, but also in adaptively moving content as application characteristics change over time. Operating system and/or runtime level solutions based on artificial intelligence (AI) and machine learning (ML) that optimize memory allocations and data movement by transparently mapping application behavior to the underlying hardware are therefore highly desired.

Although a large body of existing work explores various ML approaches for heterogeneous memory management [12,18,43,44], to the best of our knowledge none of this work applies nonsupervised learning such as reinforcement learning (RL) [41]. This gap exists despite RL's enormous potential that has been demonstrated in a wide range of fields recently [31]. RL evolves an agent to refine its policy through repeatedly interacting with the environment. Hence it requires rapid and low-overhead estimation of application execution time as a function of memory layout over heterogeneous memory devices. Cycle-level full-system simulators such as gem5 [8] and cycle-accurate memory simulators such as Ramulator [25] and NVSIM [11] incur slowdowns that are prohibitive for such a scenario. Additionally, restricting the simulation to memory devices only, namely by feeding memory access traces (captured by tools such as PIN [28] or DynInst [9]) into memory simulators, loses timing information about the computation, in turn degrading the accuracy of the overall simulation. Furthermore, these tools are still orders of magnitude slower than execution on real hardware.

This paper explores an alternative approach to rapid execution time estimation over heterogeneous memory devices, a method we call *differential tracing*. The basic idea is to obtain high-fidelity memory access traces running on real hardware using different memory devices; matching the traces to identify differences in runtime; and, based on this information, providing an estimate for execution time as a function of the virtual memory to device mapping. To this end, we utilize Intel's precise event-based sampling (PEBS) [20] mechanism and propose a number of extensions (e.g., the notion of *application phasemarks*) to the tracing mechanism that enables high-accuracy matching of memory traces. Using the matched traces, we develop an estimator that provides a runtime estimate substantially faster than cycle-level simulators.

Specifically, in this paper we make the following contributions.

- We address the issue of providing an execution time estimator for hybrid memory systems without incurring unacceptable slowdowns that would otherwise be prohibitive in iterative machine learning methods such as RL.

- We introduce a number of novel extensions to sampling-based memory access tracing (e.g., application phasemarks) that improve our ability to match memory traces.
- We evaluate our proposal on four HPC miniapplications across a wide range of memory layouts and compare the estimates with real hardware execution.

We find that the proposed method provides an average estimation error of 4.4% compared with execution on real hardware, while it runs orders of magnitude faster than gem5 and Ramulator.

The rest of the paper is organized as follows. We begin with further motivation in Sect. 2. Section 3 provides background information on memory access tracing and lightweight kernels. Our custom PEBS driver and the estimator are detailed in Sect. 4, and evaluation is provided in Sect. 5. Section 6 provides additional discussion, Sect. 7 surveys related work, and Sect. 8 concludes the paper.

2 Motivation

Before getting into the details of our proposal, we provide a high-level overview of the approach we are pursuing. Our aim is to further clarify the motivation for this work. Figure 1 outlines the idea of RL-based heterogeneous memory management.

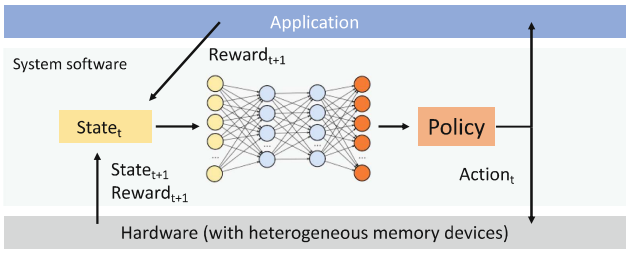


Fig. 1. Reinforcement-learning-based heterogeneous memory management.

In essence, the system software runs an RL agent that periodically observes application behavior through low-level hardware metrics such as memory access patterns, the current utilization of memory bandwidth, and the measured arithmetic intensity. Subsequently, it feeds this state information into a policy network that infers an action for potentially rearranging the memory layout of the application, that is, moving data across memory devices. In turn, the application (optionally in cooperation with the hardware) provides feedback on progress in the form of rewards, for example, inverse proportionally with execution time. The agent's goal is to maximize rewards and thus to minimize execution time.

Ideally, one would train such agents in a real execution environment on actual hardware. However, RL requires a large number of iterations for exploring

the environment, which renders real-hardware-based training extremely resource demanding. Therefore, a better approach is to train the agent offline with a surrogate hardware model faster than the actual hardware. In the remainder of the paper, we call this model an *estimator*. While existing hardware simulators can provide accurate runtime estimation, they are impractical because of the immense slowdown they incur (see Sect. 5 for a quantitative characterization of the overhead). Instead, what we need is a simulation environment that provides swift estimation of application execution time as a function of the memory layout.

In summary, we emphasize that the goal of this study is not to optimize the memory layout of the particular applications considered for evaluation but, rather, to provide a simulation environment that can be used to train machine learning models for memory management in a general context.

3 Background

3.1 Precise Event-Based Sampling

PEBS is a feature of some Intel microarchitectures that builds on top of Intel's Performance Counter Monitor (PCM) facility [20]. PCM enables the monitoring of a number of predefined processor performance counters by monitoring the number of occurrences of the specified events[1] in a set of dedicated hardware registers.

PEBS extends the idea of PCM by transparently storing additional processor information while monitoring a PCM event. However, only a small subset of the PCM events actually support PEBS. A "PEBS record" is stored by the CPU in a user-defined buffer when a configurable number of PCM events, named the "PEBS reset", occur. The actual PEBS record format depends on the microarchitecture, but it generally includes the set of general-purpose registers as well as the virtual address for load/store operations.

A *PEBS assist* in Intel nomenclature is the action of storing the PEBS record into the CPU buffer. When the number of records written by the PEBS assist events reaches a configurable threshold inside the PEBS buffer, an interrupt is triggered. The interrupt handler is expected to process the PEBS data and clear the buffer, allowing the CPU to continue storing more records. The smaller the threshold, the more frequent the interrupt requests (IRQs). We note that the PEBS assist does not store any timing information. Timestamping the PEBS data, however, can potentially occur in the IRQ handler.

3.2 Lightweight Kernel-Based Development Environment

Lightweight multikernels have emerged as an alternative operating system architecture for HPC, where the basic idea is to run Linux and a lightweight kernel (LWK) side-by-side in compute nodes to attain the scalability properties of

[1] The exact availability of events depends on the processor's microarchitecture.

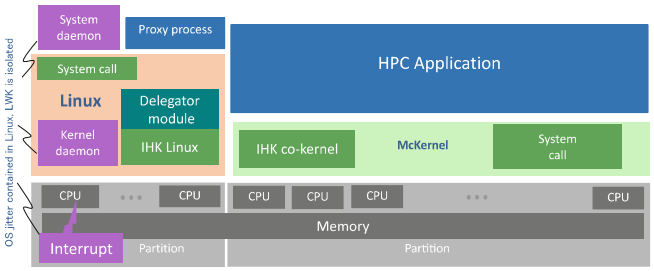


Fig. 2. Overview of the IHK/McKernel architecture.

LWKs and full compatibility with Linux at the same time. IHK/McKernel is a multikernel OS whose architecture is depicted in Fig. 2. A low-level software infrastructure, called Interface for Heterogeneous Kernels (IHK) [40], provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory), and it enables management of lightweight kernels. IHK can allocate and release host resources dynamically. No reboot of the host machine is required when altering its configuration, thus enabling relatively straightforward deployment of the multikernel stack on a wide range of Linux distributions.

McKernel is a lightweight co-kernel developed on top of IHK [15]. It is designed explicitly for HPC workloads, but it retains a Linux-compatible application binary interface so that it can execute unmodified Linux binaries. McKernel implements only a small set of performance-sensitive system calls; the rest of the OS services are delegated to Linux. Specifically, McKernel provides its own memory management, it supports processes and multithreading, it has a simple round-robin cooperative (tickless) scheduler, and it implements standard POSIX signaling. It also implements interprocess memory mappings, and it offers interfaces for accessing hardware performance counters.

McKernel has a number of favorable properties with respect to this study. First, it is highly deterministic. Not only does it provide predictable performance across multiple executions of the same program, but it also ensures that the same virtual memory ranges are assigned to a process when executed multiple times, assuming that the application itself is deterministic. As we will see, this significantly simplifies comparing memory access traces obtained from multiple executions.

Second, McKernel's relatively simple source code provides fertile ground for developing custom kernel-level solutions. For example, it provides a custom PEBS driver [29] that we extend with an API to capture higher-level application information (e.g., the application phasemarks discussed in Sect. 4.1), as well as another custom interface that enables selectively binding parts of the application address space to specific memory devices without changing the application code (detailed in Sect. 4.2).

4 Design and Implementation

This section discusses the design of our proposed execution time estimator along with its most relevant implementation details.

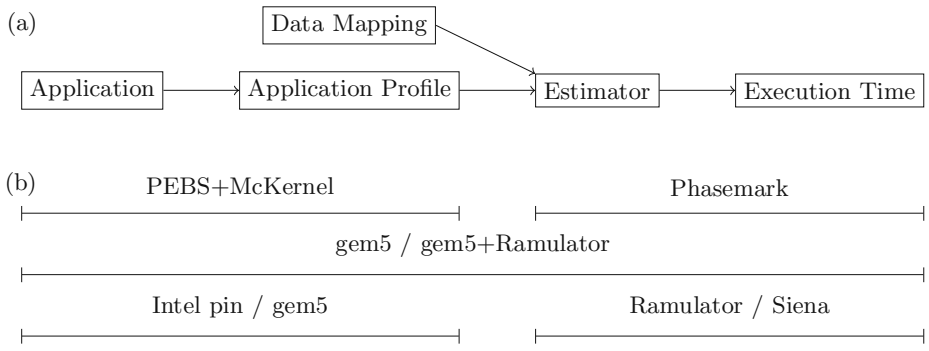


Fig. 3. High-level representation of the steps needed to build the proposed estimator (a) and the tools that assist with the implementation (b).

Figure 3(a) gives an overview of the steps to build the proposed estimator. The system comprises two main pieces: the application profiler and the estimator. First, we collect memory access traces of the target application (*Application Profile* in the figure). Since we want to train an agent offline, trace collection may be slow for the purpose of training. However, the profiling step must have a low overhead once the agent is deployed (i.e., during inference), and the estimator has to be fast for training. Thus, it is desired that both pieces be fast, incur low overhead, and attain high accuracy.

In our implementation of this system (*PEBS+McKernel* in Fig. 3(b)), we collect high-frequency memory access traces from a real, heterogeneous memory-equipped hardware environment where we place application content into different memory devices. Therefore, the application profile is composed of sampled memory access traces annotated with timing information, once for each memory device of the target computing system. Using sampling hardware counters is effectively the lowest-overhead, application-oblivious way to collect an application's memory access trace.

The estimator, which we will describe in more detail below, matches the traces and identifies execution phases (*Phasemark* in the figure) along with the accessed memory regions that impact performance. Taking into account the discrepancy between traces from different memory devices, it estimates execution time based on input that describes the layout of application data with respect to the underlying memory devices, the mapping between virtual memory ranges to the corresponding memory devices that back those mappings (*Data Mapping* in Fig. 3(a)).

Different approaches exist for implementing such a system, as outlined in Fig. 3(b). It can be implemented with a different profiling method and/or a different estimator (*Intel pin/gem5* and *Ramulator/Siena* in the figure, respectively) or even combining the profiling and estimation steps into a single step (*gem5/gem5+Ramulator*). We found that existing approaches are impractical in the context of reinforcement learning because RL requires both low-overhead profiling for the inference step and a fast estimator for the training step. We evaluate some of these approaches in Sect. 5.

We first describe the details of our memory access tracing mechanism.

4.1 Memory Access Tracing and Application Phasemarks

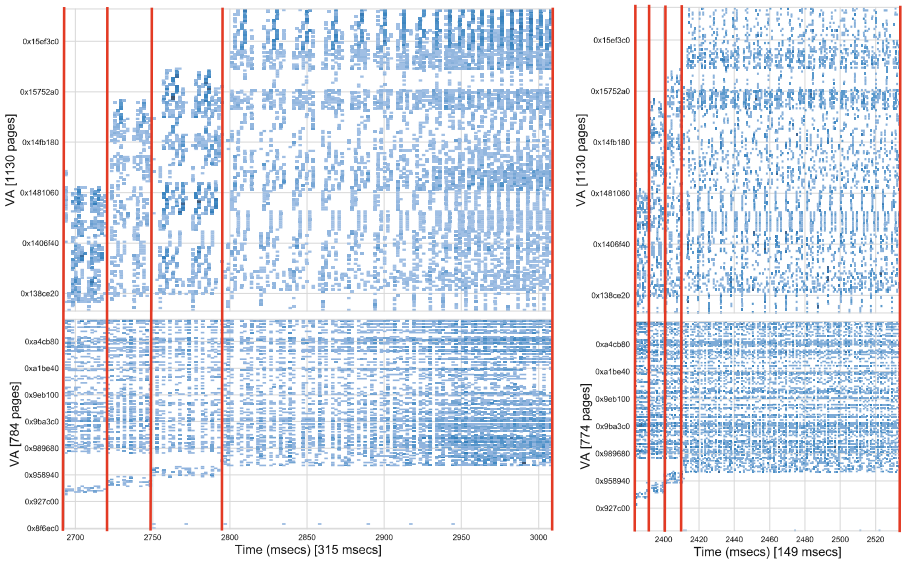
To track application-level memory accesses, we utilize Intel's PEBS facility. Specifically, we configure PEBS on the event of last-level cache misses for which the PEBS records include not only the set of general-purpose registers but also the virtual address for the particular load/store operation that triggered the cache miss, effectively capturing the memory access pattern of the application.

It has been reported previously that standard PEBS drivers incur nontrivial overhead and have limited configuration flexibility [1,26,30]. For example, in both the Linux kernel's PEBS driver and the one provided by Intel's vTune software, no interface is available for controlling the internal PEBS assist buffer size, which implicitly controls the frequency of PEBS interrupts that enable the annotation of PEBS records with high-granularity timestamps. Olson et al. also reported that decreasing the PEBS reset value below 128 on Linux caused the system to crash [30]. For these reasons we utilize McKernel's custom PEBS driver, which has been shown to have negligible runtime overhead even at very high-granularity tracing, for example, by capturing memory accesses with a PEBS reset counter as low as 16 [29].

In addition to high-frequency tracing, we extend the kernel device driver to annotate PEBS records with two extra pieces of information. First, we introduce the notion of *application phases*, for which we add a dedicated `phasemark()` system call in McKernel. The call simply increments a counter in the PEBS driver, which is in turn appended to each PEBS record. Second, we automatically record the number of retired instructions elapsed since the beginning of the last application phase, which again is attached to the PEBS record. As we will see below, this extra information enables us to match memory access traces from different memory devices with very high accuracy. We note that phasemark calls can be inserted into the application source code either manually or through compiler-level code transformation.

Figure 4 highlights the impact of phasemarks in two memory access traces captured from DDR4 and high-bandwidth memory, respectively, when running the Lulesh miniapplication [23]. For more information on the hardware platform used for this experiment as well as on the specifics of how we execute the application, see Sect. 5. The x-axis of the figures indicates elapsed time, while the y-axis shows virtual page indices (i.e., virtual addresses divided by the page size).

The width of the two plots is proportional to the execution time, while the red vertical lines pinpoint application phasemarks captured by the PEBS driver. The two plots show the same four phases of the application, with the only difference being that the application was running on different memory devices.



(a) DDR4 (b) MCDRAM

Fig. 4. Lulesh memory traces from DDR4 vs. MCDRAM, annotated with application phasemarks.

As shown, the virtual memory ranges of the two executions are almost identical. This is due to the deterministic behavior of McKernel's memory management subsystem. In addition, phasemarks help determine how much a given application phase is impacted by the fact that memory content is placed into a particular memory device. This information is especially important because not all phases experience the same effect. For example, the execution time of the fourth phase in the figure is reduced by 44% when using MCDRAM; the first phase, however, becomes almost 4× faster. Had we not marked the different phases, trace matching would become significantly more complex, since it would need to identify parts of the trace where the application proceeds at a different pace from that of others when executed out of a different memory device. In contrast, with the presence of phasemarks, we have stable anchors for periodic synchronization while processing the traces. In Sect. 5 we quantitatively characterize the impact of phasemarks on runtime estimation accuracy.

4.2 Execution Time Estimation and Verification

Estimation. The mechanism of the execution time estimator is remarkably simple. The algorithm processes memory traces of a given application obtained from different memory devices by iterating through the individual phases supplemented by the phasemark annotation. In a given phase, the memory access traces are further divided into windows based on the number of retired instructions associated with the memory access samples. Much to our surprise, we observe some discrepancy between the number of retired instructions (associated with particular phases) captured by the PEBS driver based simply on which underlying memory device is utilized. We are unsure whether this is due to some timing effect caused by the difference between the memory devices or an issue with performance counter implementation in the CPU. Either way, to guarantee that a given phase is processed at the same pace from both traces, we configure the window lengths proportionally. The window length is a parameter of the estimator, and we typically configure it to cover a few hundred thousand instructions according to the baseline trace.

In a given window, the estimator iterates the traces and records the number of accesses that hit each particular memory device according to the mapping between the virtual memory ranges and the backing devices. Based on the ratio of the number of accesses, we calculate the execution time of the given window by skewing it proportionally between the measured times over different devices, e.g., for a DRAM plus HBM system we use the following formula: $t_{est} = t_{DRAM} - (t_{DRAM} - t_{HBM}) \cdot \frac{\#accesses_{HBM}}{\#accesses_{all}}$.

As one may notice, this mechanism completely disregards data dependencies among memory accesses and greatly simplifies the interpretation of memory access traces. Nevertheless, as we will see in Sect. 5, this simple approach (in combination with phasemarks) proves to be surprisingly accurate. We also note that utmost accuracy is not required for the ML training process to be successful; rather, it is sufficient if it is expressive enough to guide the learning algorithm to the right optimization path.

Verification. To verify the accuracy of the estimator, we extend McKernel's memory management code with two custom APIs. One allows the specification of a list of virtual memory ranges along with their target memory device; the other makes it possible to indicate a percentage that is interpreted as the fraction of application pages that are to be mapped to a given memory device. The kernel automatically places the memory of the calling process on the target device irrespective of whether it covers the stack, heap, data/BSS sections, or anonymous memory mappings in the process's address space.

As opposed to standard POSIX calls such as `set_mempolicy()` or `mbind()` that need to be invoked at the application level, this memory placement mechanism is carried out in an application-transparent fashion. This approach greatly simplifies experimentation because we do not need to make modifications to individual applications. Using the APIs, we can easily verify the accuracy of the proposed estimator against measurements on real hardware.

5 Evaluation

All of our experiments were performed on an Intel® Xeon Phi™ 7250 Knights Landing (KNL) processor, which consists of 68 CPU cores, accommodating 4 hardware threads per core. The processor provides 16 GB of integrated, high-bandwidth MCDRAM, and it is accompanied by 96 GB of DDR4 RAM. We configured the KNL processor in Quadrant flat mode; in other words, MCDRAM and DDR4 RAM are addressable at different physical memory locations. We used 64 CPU cores for applications and reserved the rest for OS activities. While we acknowledge that the KNL platform has come of age, we emphasize that our proposal is orthogonal to the underlying hardware. We use KNL because it is currently the only generally available CPU architecture supporting both high-bandwidth memory and regular DDR4. Note that Intel has already announced its upcoming Sapphire Rapids CPU model that will provide a similar hybrid memory environment [3]. For the wall-clock measurements of the estimator, we use an Intel® Xeon™ Platinum 8280 (Cascade Lake) CPU equipped platform.

5.1 Application Benchmarks

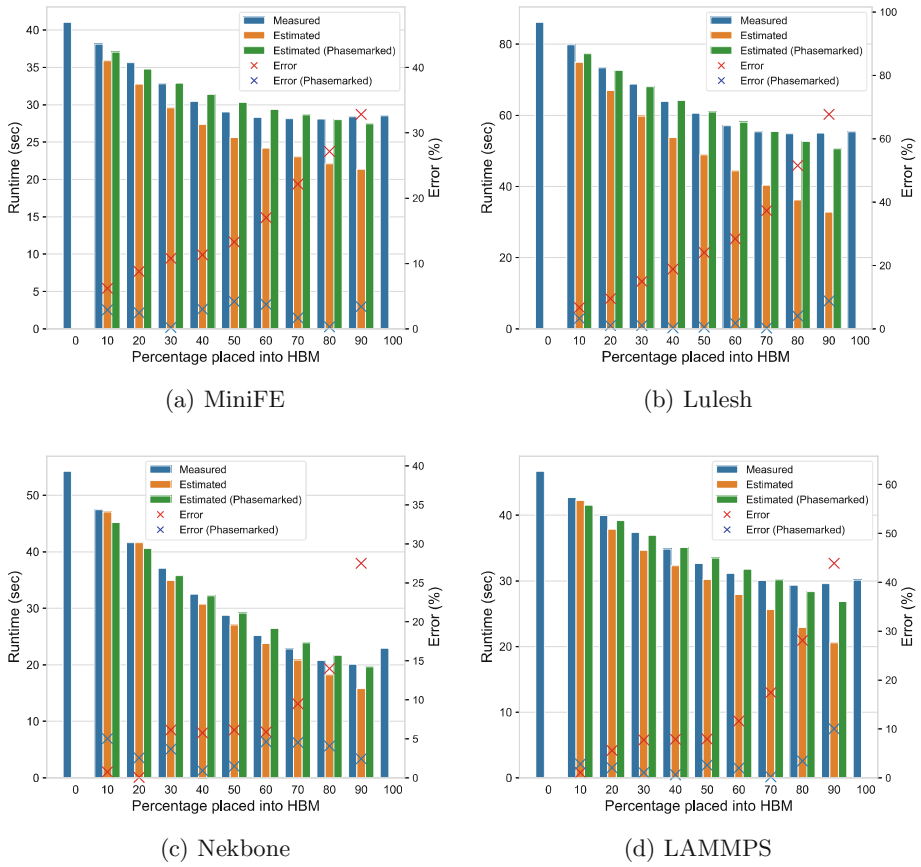
To evaluate the proposed estimator, we chose the following miniapplications primarily because they are the subject of a substantial runtime difference when executed out of high-bandwidth memory.

- **MiniFE** is a proxy application for unstructured implicit finite element codes. It is similar to HPCCG and pHPCCG but provides a much more complete vertical covering of the steps in this class of applications [17].
- **Lulesh** is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics benchmark, which is part of the Shock Hydrodynamics Challenge Problem. It was originally defined and implemented by Lawrence Livermore National Laboratory, and it is a widely studied proxy application in U.D. Department of Energy co-design efforts [23].
- **LAMMPS** is an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator. LAMPPS is a classical molecular dynamics code [36].
- **Nekbone** solves a standard Poisson equation using a conjugate gradient iteration with a simple preconditioner on a block or linear geometry. Nekbone exposes the principal computational kernel that is pertinent to Nek5000 [5].

All our measurements are performed in flat MPI configuration, that is, running 64 MPI ranks on a single node with a dedicated CPU core for each process. This setup enables us to achieve two important goals. First, we make sure that we exercise the entire chip and measure a practical application deployment. Second, the single-threaded execution of each rank ensures deterministic behavior with respect to memory mappings, which in turn enables us to easily measure configurations where only specific ranges of the address space are mapped to high-bandwidth memory. We also note that we observe negligible performance variation across multiple executions on McKernel, and thus we omit error bars on measured data points. As for PEBS, we configure the reset value to 16.

5.2 Results

We provide two sets of experiments with respect to estimation accuracy. In the first setup we gradually increase the fraction of the application address space that is mapped to high-bandwidth memory from 0% (i.e., running entirely out of DDR4) all the way up to 100%, where all memory is allocated out of MCDRAM. We increase the ratio in steps of 10%. Figure 5 summarizes the results.



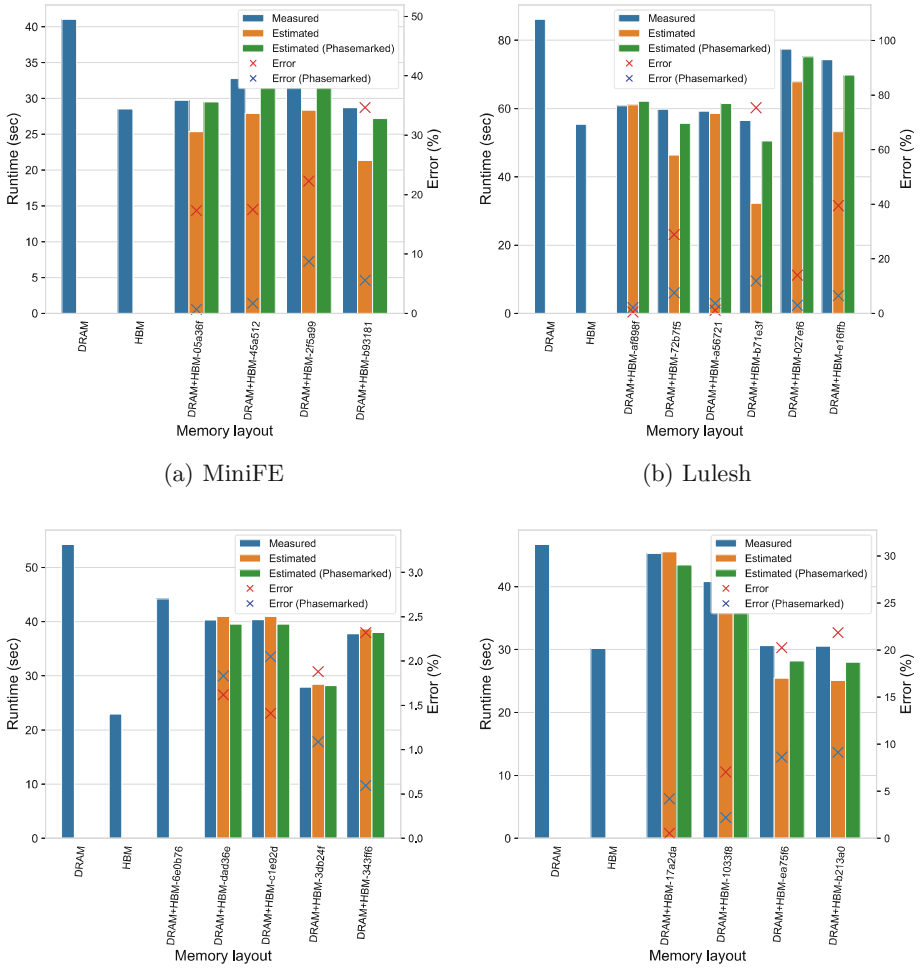
(a) MiniFE (b) Lulesh

(c) Nekbone (d) LAMMPS

Fig. 5. Runtime estimations vs. measurements as a function of data fraction placed in high-bandwidth memory.

On each plot the x-axis indicates the fraction of application memory that is mapped to HBM. The left y-axis shows execution time, where the blue, orange and green bars indicate runtimes as measured, estimated w/o phasemarks, and estimated with phasemarks, respectively. We do not estimate values for full DDR4 and MCDRAM executions. The right y-axis covers estimation

error (against the measured values). The actual values are shown by the blue and red crosses, for with and w/o phasemarks, respectively.



(a) MiniFE (b) Lulesh

(c) Nekbone (d) LAMMPS

Fig. 6. Runtime estimations vs. measurements as a function of data ranges placed in high-bandwidth memory.

As shown, without using phasemarks we endure estimation errors up to over 60%, while with the incorporation of phasemark information the proposed mechanism provides a remarkably accurate estimation of runtimes. Specifically, the average estimation error when using phasemarks for MiniFE, Lulesh, Nekbone, and LAMMPS is 2.4%, 2.3%, 3.2%, and 2.7%, respectively. The largest error (while using phasemarks) we observe across all the experiments is for LAMMPS

at 90% of the memory placed into HBM where we see approximately a 10% error. We find this particular result counterintuitive because most pages are in HBM already; we are still investigating the root cause of it.

The second set of experiments covers arbitrary ranges placed into high-bandwidth memory. We emphasize that the importance of these experiments lies in the fact that they mimic the conditions the estimator would encounter during RL training. Results are shown in Fig. 6.

The plots are similar to those in Fig. 5 except for the x-axis. For brevity we omit listing the actual address ranges (also because they do not carry any particular meaning), and we use short notations on the x-axis to indicate different configurations where select memory ranges are placed into MCDRAM. We handpicked these ranges by visually examining traces and by algorithmically identifying areas where a large number of the accesses concentrate.

Again, without using application phasemarks we observe errors up to 75%. To the contrary, when utilizing phasemarks the average estimation error for MiniFE, Lulesh, Nekbone, and LAMMPS is 4.1%, 5.7%, 3.1%, and 6%, respectively. We note that although in a few cases the error using phasemarks exceeds that of without it (e.g., in Fig. 6c), the error is already very small in these cases. While these numbers are somewhat elevated compared with those of the more regular percentage-based experiments, we believe these are still well within the acceptable range for driving ML training. Overall, across all experiments, the estimator with phasemarks yields an average error rate of 4.4%.

Estimation Time. As depicted in Fig. 3, the profiling and estimation steps proposed in this paper may be compared with other methods having similar utility, that is, to estimate the application execution time as a function of the mapping between application data and physical memory. Here, we compare the overhead of our method with that of three other methods.

Table 1. Comparison of actual application runtime, the wall-clock time of the proposed estimator, and simulation times of Ramulator and gem5.

Application	Runtime (measured)	Estimator (measured)	Ramulator (estimated)	gem5 (estimated)
MiniFE	41 s	∼9 s	∼1 h	∼14 days
Lulesh	86 s	∼54 s	∼2 h	∼29 days
Nekbone	54 s	∼66 s	∼1 h	∼18 days
LAMMPS	46 s	∼39 s	∼1 h	∼15 days

Based on measurements, we report the upper limit of application runtime (i.e., running out of DRAM) and the wall-clock time it takes the proposed mechanism to give an estimate. In addition, we estimate the simulation times

of Ramulator and gem5. Ramulator [25] is a memory simulator capable of simulating multiple memory technologies. In our experience, using Ramulator to process a memory access trace obtained with the Intel Pin [28] binary instrumentation tool was about 100 to 1,000 times slower than running the actual application. We estimate wall-clock times based on these values. The gem5 [8] simulator is a cycle-level simulator, modeling several components of a hardware platform including CPUs, caches, and the memory hierarchy. We estimate gem5 runtimes based on the approximate 30,000 times slowdown reported by Sandberg *et al.* [38], which is also in line with our own experience running smaller benchmarks.

Results are shown in Table 1. As seen, the runtimes of both gem5 and Ramulator are prohibitive for our purpose. In contrast, the proposed estimator provides runtime estimates several orders of magnitude faster. In fact, except for Nekbone, it runs faster than the application itself. We note that the slowdown in Nekbone is related to the large number of memory accesses that impacts the speed of the simulation. We leave further performance optimization of the estimator for future work. Nevertheless, we point out that the estimator runs on a single CPU core as opposed to the application that occupies at least an entire chip. Taking into account RL's ability to utilize multiple agents concurrently, our solution provides efficiency improvements proportional to the number of CPU cores even if compared with actual application runs. Moreover, since the application profile used in the proposed mechanism is based on sampled memory access traces, we can adjust the trade-off between the trace resolution and the estimator accuracy to speed up the profiling and estimation.

6 Discussion

This section provides additional discussion on various aspects of the proposal.

The ultimate goal of this study is to train ML agents that will guide memory placement in heterogeneous memory environments in an application transparent fashion. Phasemarking is used exclusively for building the environment to train the agent (i.e., for generating training data) and the expectation is that ML agents will generalize enough to work on unseen access patterns.

We emphasize that at the time of deployment an RL agent only needs to observe memory accesses (e.g., through PEBS) and there is no need for phasemarking each application when the system is deployed. Furthermore, our special-purpose OS is only utilized for the creation of training data. Once an RL agent is trained, it can be deployed in any standard OS/runtime environment with the only requirement for being able to sample memory accesses.

One might recognize the possibility to directly utilize phasemark information for memory management. While this may be feasible, it is outside the scope of this study and we leave it for future exploration. Our goal is to derive an application-transparent solution that does not require code instrumentation.

7 Related Work

A significant amount of research has been done over the years on improving page placement for complex memory architectures.

Focusing on modern memory architectures and profiling-based methods, we identify several works of interest. The first set of research studies can be characterized by the focus on designing a system or runtime using different memory tiers as a stage-in/stage-out cache. These studies can predict which pages of memory should be migrated from large and slow to small and fast memory devices ahead of time, either in a system with NVRAM and DRAM or a system with DRAM and HBM. We highlight the works of Doudali *et al.* [12–14] that showcase ML methods to predict which pages to migrate next or how often to perform migration. These works are difficult to adapt to our objective, however, since they operate under the assumption that any application page would benefit from being in fast memory at the right time, which is not necessarily the case in the HPC context [33,37]. Indeed, we aim here to select the right placement for each page and not to design a paging scheme that would move the working set of an application in and out of a hierarchy of devices. We also note that many of the above-mentioned studies consider only single CPU core execution, which we think is unrealistic in an HPC setting. We can also differentiate these works with respect to the profiling method used: whether it is based on estimating locality metrics (e.g., reuse distance) [2,12,24] or a form of memory pressure (e.g., access count per region) [6,30,39].

We further highlight studies providing heuristics or software facilities for data migration between heterogeneous memories [7,35], either through the use of the same metrics as above or through more knowledge of the application. Phase detection is also an extensive field of study, surveyed in [10]. We note that most phase detection methods, in particular architecture-supported ones, tend to be used for reconfiguration purposes (make a change in a policy) and not as much for comparison of traces of the same application in different setups. Nevertheless, we will investigate the use of other lightweight phase detection systems in our future work.

Binary instrumentation tools can also be used to track memory accesses, filter them between the last-level cache and memory, and model the timing of the instructions. Such solutions could be used as profilers for our trace-based estimator. However, the overhead of binary instrumentation-based methods for memory analysis tools has been shown to increase the number of instructions to execute by 10 times [42]. Intel Pin [28] (3.21) is a binary instrumentation framework shipped with a single-level cache emulator tool. Although it could be used here as a profiler, in our experience the overhead of the tool is more on the order of 100 to a 1000 times.

Ramulator [25], the memory simulator we used for evaluation, can also be combined with other tools such as Siena [34] to simulate heterogeneous memory systems. These tools can be used as an estimator to evaluate the impact of data mapping on applications in a fashion similar to our own estimator. Unlike with

our tool, however, where we can rely on sampled memory access traces, these tools require a complete trace to provide an accurate timing estimation.

8 Conclusion and Future Work

As architectural complexity grows in HPC systems, it becomes increasingly challenging to efficiently utilize these platforms. Therefore, intelligent application-transparent solutions are greatly desired. In particular, ML/AI techniques that can discover solutions without labeled data may come to the rescue. However, techniques such as reinforcement learning require a large number of interactions with the environment; and thus, when applied to heterogeneous memory management, they require rapid execution time estimation of the application running on a hybrid memory platform.

This paper has proposed a novel execution time estimation mechanism that relies on comparing sampled memory access traces obtained on real hardware from different memory devices. This relatively simple mechanism achieves remarkably accurate runtime predictions (with an average error rate of 4.4%) while running orders of magnitudes faster than high-fidelity architectural simulators. Thus, the proposed mechanism opens up the opportunity to be deployed in nonsupervised machine learning frameworks such as in RL.

Our immediate future work entails integrating the proposed estimator into an RL framework to explore the feasibility of its application to heterogeneous memory management. With respect to gem5, while it is not a suitable solution for high-speed and low-overhead runtime estimation, we intend to use it in the future as a validation platform for new architectures.

Acknowledgment. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This research was also supported by the JSPS KAKENHI Grant Number JP19K11993.

References

1. Akiyama, S., Hirofuchi, T.: Quantitative evaluation of Intel PEBS overhead for online system-noise analysis. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017 (2017)
2. Alvarez, L., Casas, M., Labarta, J., Ayguade, E., Valero, M., Moreto, M.: Runtime-guided management of stacked DRAM memories in task parallel programs. In: Proceedings of the 2018 International Conference on Supercomputing (2018)
3. AnandTech: Intel to launch next-gen Sapphire Rapids Xeon with high bandwidth memory (2021). https://www.anandtech.com/show/16795/intel-to-launch-next-gen-sapphire-rapids-xeon-with-high-bandwidth-memory
4. Angel, S., Nanavati, M., Sen, S.: Disaggregation and the Application. USENIX Association, Berkeley (2020)

5. Argonne National Laboratory: Proxy-apps for thermal hydraulics (2021). https://proxyapps.exascaleproject.org/app/nekbone/
6. Arima, E., Schulz, M.: Pattern-aware staging for hybrid memory systems. In: International Conference on High Performance Computing (2020)
7. Benoit, A., Perarnau, S., Pottier, L., Robert, Y.: A performance model to execute workflows on high-bandwidth-memory architectures. In: Proceedings of the 47th International Conference on Parallel Processing (2018)
8. Binkert, N., et al.: The gem5 simulator. SIGARCH Comput. Archit. News (2011). https://doi.org/10.1145/2024716.2024718
9. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. (2000), https://doi.org/10.1177/109434200001400404
10. Dhodapkar, A.S., Smith, J.E.: Comparing program phase detection techniques. In: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36 (2003)
11. Dong, X., Xu, C., Xie, Y., Jouppi, N.P.: NVSim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst. **31**, 994–1007 (2012)
12. Doudali, T.D., Blagodurov, S., Vishnu, A., Gurumurthi, S., Gavrilovska, A.: Kleio: A hybrid memory page scheduler with machine intelligence. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (2019)
13. Doudali, T.D., Zahka, D., Gavrilovska, A.: The case for optimizing the frequency of periodic data movements over hybrid memory systems. In: The International Symposium on Memory Systems (2020)
14. Doudali, T.D., Zahka, D., Gavrilovska, A.: Cori: dancing to the right beat of periodic data movements over hybrid memory systems. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2021)
15. Gerofi, B., Takagi, M., Hori, A., Nakamura, G., Shirasawa, T., Ishikawa, Y.: On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2016
16. Hady, F.T., Foong, A., Veal, B., Williams, D.: Platform storage performance with 3D XPoint technology. In: Proceedings of the IEEE (2017)
17. Heroux, M.A., et al.: Improving performance via mini-applications. Tech. rep, Sandia National Laboratories (2009)
18. Hildebrand, M., Khan, J., Trika, S., Lowe-Power, J., Akella, V.: AutoTM: automatic tensor movement in heterogeneous memory systems using integer linear programming. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (2020). https://doi.org/10.1145/3373376.3378465
19. HMC Consortium: Hybrid Memory Cube Specification 2.1. (2015). http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf
20. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer Manuals (2021). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
21. JEDEC Solid State Technology Association: High Bandwidth Memory (HBM) DRAM (2015)
22. JEDEC Solid State Technology Association: Graphics Double Data Rate 6 (GDDR6) SGRAM standard (2017)

23. Karlin, I., Keasler, J., Neely, R.: LULESH 2.0 updates and changes. Tech. rep., Lawrence Livermore National Laboratory (2013)
24. Kim, J., Choe, W., Ahn, J.: Exploring the design space of page management for multi-tiered memory systems. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21) (2021)
25. Kim, Y., Yang, W., Mutlu, O.: Ramulator: a fast and extensible DRAM simulator. IEEE Comput. Archit. Lett. **15**, 45–49 (2016)
26. Larysch, F.: Fine-grained estimation of memory bandwidth utilization. Master's thesis (2016)
27. Lee, B.C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable DRAM alternative. SIGARCH Comput. Archit. News (2009). https://doi.org/10.1145/1555815.1555758
28. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
29. Nonell, A.R., Gerofi, B., Bautista-Gomez, L., Martinet, D., Querol, V.B., Ishikawa, Y.: On the applicability of PEBS based online memory access tracking for heterogeneous memory management at scale. In: Proceedings of the Workshop on Memory Centric High Performance Computing (2018)
30. Olson, M.B., Zhou, T., Jantz, M.R., Doshi, K.A., Lopez, M.G., Hernandez, O.: MemBrain: automated application guidance for hybrid memory systems. In: IEEE International Conference on Networking, Architecture, and Storage (2018)
31. Padakandla, S.: A survey of reinforcement learning algorithms for dynamically varying environments. ACM Comput. Surv. **54**(6) (2021). https://doi.org/10.1145/3459991
32. Park, K.-T., et al.: 19.5 three-dimensional 128Gb MLC vertical NAND flash-memory with 24-WL stacked layers and 50MB/s high-speed programming. In: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC) (2014)
33. Parsons, B.S.: Initial benchmarking of the Intel 3D-stacked MCDRAM. Tech. rep, ERDC (2019)
34. Peng, I.B., Vetter, J.S.: Siena: exploring the design space of heterogeneous memory systems. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (2018)
35. Peng, I.B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., Markidis, S.: RTHMS: a tool for data placement on hybrid memory system. ACM SIGPLAN Notices **52**, 82–91 (2017)
36. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. J. Comput. Phy. **117**, 1–19 (1995)
37. Pohl, C.: Exploiting manycore architectures for parallel data stream processing. In: Grundlagen von Datenbanken, pp. 66–71 (2017)
38. Sandberg, A., Diestelhorst, S., Wang, W.: Architectural exploration with gem5 (2017). https://www.gem5.org/assets/files/ASPLOS2017_gem5_tutorial.pdf
39. Servat, H., Peña, A.J., Llort, G., Mercadal, E., Hoppe, H.C., Labarta, J.: Automating the application data placement in hybrid memory systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER) (2017)
40. Shimosawa, T., et al.: Interface for heterogeneous kernels: A framework to enable hybrid OS designs targeting high performance computing on manycore architectures. In: 21st International Conference on High Performance Computing (2014)
41. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (1998). http://www.cs.ualberta.ca/~sutton/book/the-book.html

42. Uh, G.R., Cohn, R., Yadavalli, B., Peri, R., Ayyagari, R.: Analyzing dynamic binary instrumentation overhead. In: WBIA Workshop at ASPLOS. Citeseer (2006)
43. Wu, K., Ren, J., Li, D.: Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (2018)
44. Yu, S., Park, S., Baek, W.: Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In: Proceedings of the International Conference on Supercomputing, pp. 1–10 (2017)
45. Zambelli, C., Navarro, G., Sousa, V., Prejbeanu, I.L., Perniola, L.: Phase change and magnetic memories for solid-state drive applications. In: Proceedings of the IEEE (2017)