Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Bewertung und Vergleich von Datenplatzierungs-Optimierungswerkzeugen für heterogene Speicherarchitekturen

Evaluating and Comparing Data Placement Optimization Frameworks for Heterogeneous Memory Systems

Bachelorarbeit

Ben-Jay Huckebrink
Matrikelnummer: 445219

Aachen, den 29. September 2025

Communicated by Prof. Dr. Matthias S. Müller

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Dr. rer. nat. Stefan Lankes (*)
Betreuer:	Dr. rer. nat. Jannis Klinkenberg (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Automation of Complex Power Systems, RWTH Aachen University

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 29. September 2025

Kurzfassung

Die speicherbezogenen Anforderungen von wissenschaftlichen Anwendungen steigen in immer höherem Tempo. Der traditionell genutzte Hauptspeicher, *Dynamic Random Access Memory* (DRAM), hält mit diesen steigenden Kapazitäts-, Geschwindigkeits- und Energieeffizienzanforderungen nicht mit. Daher werden *heterogene Speicherarchitekturen*, welche mehrere Speichertypen wie z.B. *Non-Volatile Memory* (NVM) und *High-Bandwidth Memory* (HBM) neben DRAM nutzen, immer präsenter. Um die Vorteile solcher Architekturen zu nutzen, werden einzelne Datenstrukturen der Anwendungen je nach deren *Speicherzugriffsmustern* in verschiedene Speichertypen platziert. Da eine solche Datenplatzierungs-Optimierung manuell vorzunehmen viel Wissen über die Anwendung und viel Zeit erfordert, wurden *Datenplatzierungs-Optimierungswerkzeuge* entwickelt, um diesen Prozess zu automatisieren und die getroffenen Entscheidungen zu verbessern. Jedoch hat die Forschung an diesen Werkzeugen deren Wirksamkeit nicht ausreichend untersucht. Zumeist wird nur die ausführungszeitbezogene Leistung der Platzierungsentscheidungen getestet, nicht aber die Benutzerfreundlichkeit der Werkzeuge oder deren Energieeffizienzvorteile. Zudem vergleicht diese existierende Forschung die verschiedenen Werkzeuge nicht miteinander. Zusammengenommen behindern diese Unzulänglichkeiten die weitere Forschung an solchen Werkzeugen, da die *spezifischen* Stärken und Schwächen von bereits existierenden Ansätzen unbekannt bleiben und daher deren Schwächen nicht systematisch behoben werden können.

In dieser Arbeit adressiere ich diese Mängel, indem ich drei Datenplatzierungs-Optimierungswerkzeuge vergleiche und tiefergehend bewerte. Zu diesem Zweck entwickle ich einen hochgradig konfigurierbaren synthetischen Benchmark, welcher systematisch seine Speicherzugriffsmuster ändern kann. Diese Konfigurierbarkeit erlaubt mir, die spezifischen Stärken und Schwächen jedes dieser Werkzeuge zu untersuchen und deren Auswirkungen auf die Ausführungszeit und Energieeffizienz der getroffenen Platzierungsentscheidungen zu quantifizieren. Indem ich die Werkzeuge zusätzlich mit vier Proxy-Anwendungen teste, kann ich die realen Auswirkungen der identifizierten Vor- und Nachteile beurteilen. Zudem decken diese Proxy-Applikationen Schwächen in der Benutzerfreundlichkeit der Werkzeuge auf. Basierend auf meinen Ergebnissen schlage ich Modifikationen für diese Werkzeuge vor, die deren Platzierungsentscheidungen und Benutzerfreundlichkeit verbessern.

Stichwörter: Heterogene Speicherarchitekturen, Non-Volatile Memory, High-Bandwidth Memory, Datenplatzierungs-Optimierung

Abstract

The memory-related demands of scientific applications rise at an ever-accelerating pace. However, traditional dynamic random access memory (DRAM) has not kept up with these increasing memory capacity, speed, and energy efficiency demands. In response, *heterogeneous memory systems* employing multiple memory types, such as *non-volatile memory* (NVM) or *high-bandwidth memory* (HBM), alongside DRAM have risen to prevalence. Leveraging the advantages of such systems involves placing individual application data structures into different memory types depending on their *memory access behaviors*. Since manually conducting such a placement optimization requires detailed application knowledge and a large time investment, previous research developed *data placement optimization frameworks* to automate this process and improve the placement decisions made. However, previous research on these frameworks has not adequately evaluated their efficacy. Most existing work tests only the execution time performance of the frameworks' placement decisions, leaving the frameworks' user experience and energy efficiency benefits unquantified. Crucially, existing research also does not compare the different frameworks against one another. In combination, these shortcomings impede research on future frameworks, since the *specific* strengths and weaknesses of already existing approaches remain unknown, meaning their weaknesses cannot be improved systematically.

In this thesis, I address this shortage by evaluating and comparing three state-of-the-art data placement optimization frameworks in-depth. For this purpose, I develop a custom, highly configurable synthetic benchmark that can systematically alter its memory access behaviors. This configurability allows me to detail specific strengths and weaknesses of each framework's placement optimization algorithm and quantify their impact in terms of the execution time and energy efficiency the made placement decisions achieve. By also testing the frameworks on four proxy applications, I assess the real-world implications of the identified advantages and disadvantages. Further, using the proxy applications, I uncover shortcomings in the frameworks' user experience. Based on my observations, I propose modifications to the frameworks to improve their decision-making and their user experience.

Keywords: Heterogeneous memory, Non-volatile memory, High-bandwidth memory, Data placement optimization

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1. Introduction

Within High Performance Computing (HPC) systems, the gap between memory and computational performance has steadily increased. This trend and its consequences have been exacerbated vastly by applications becoming ever more bound by memory performance in recent years [41, 45, 33]. However, modern applications not only pose heightened requirements in terms of the memory's speed but also its capacity. Further, to meet wider sustainability goals, the energy consumption of HPC systems has come under increasing scrutiny, with the memory being one of the main contributors to an HPC system's energy consumption.

To address these concerns, *heterogeneous memory systems* (HMSes) have become increasingly popular in the HPC domain. These systems employ multiple, specialized memory types such as *non-volatile memory* (NVM) or *high-bandwidth memory* (HBM) alongside the traditional main memory type, *dynamic random access memory* (DRAM), to match the heterogeneous memory-related demands of modern workloads [41, 33]. Accelerators such as Graphics Processing Units (GPUs), whose usage also increases in contemporary HPC systems, further this memory heterogeneity by introducing their own memory spaces. However, in this work, I will limit my discussion to only CPU-side memory heterogeneity.

To use these heterogeneous memory systems to their fullest extent, the individual *memory objects* each application allocates must be distributed among the different memory types [41, 24, 27]. The optimal solution to this placement decision problem depends on the objects' *memory access behaviors* and the overarching optimization goal. For instance, placement decisions optimizing application performance might not yield the best energy efficiency [29]. Thus, optimizing the data placement of an application requires its programmer to have detailed knowledge of their application's memory access behaviors and the performance and energy characteristics of each memory type in their system. Furthermore, the programmer must manually decide for each of the (potentially hundreds of) objects their application allocates in which memory type to place it. Since not all programmers have such knowledge and time to manually conduct this placement optimization, previous research has developed *data placement optimization frameworks* to automate this decision process.

However, previous research on such frameworks does not sufficiently evaluate their promulgated tools and algorithms. Most existing work only tests the execution time performance of the frameworks' placement decisions by employing small sets of applications. These sets often do not systematically cover all possible memory access behaviors, meaning they are insufficient to identify specific weaknesses in the frameworks' placement optimization algorithms on their own. Further, focusing on the performance of the placement decisions alone fails to discuss the user experience of

applying these frameworks to any given application. Yet, said user experience is vital for these frameworks to see widespread use in the future. Lastly, existing work does not compare the already existing frameworks and their approaches. In combination, all the forenamed shortcomings imply that future frameworks cannot systematically improve upon existing solutions since the exact advantages and disadvantages of each of them have not been identified.

Consequently, this thesis provides an all-encompassing, in-depth evaluation and comparison of three state-of-the-art placement optimization frameworks. In my evaluation, I focus on the execution time performance of these frameworks since all three frameworks I compare seek to optimize application performance. Yet, I also quantify the secondary energy consumption benefits this performance-focused optimization entails. Additionally, I detail the user experience of each framework. For said evaluation, I use a custom-developed synthetic benchmark, which can alter its memory access behaviors systematically, thereby delivering detailed insights into the frameworks' placement optimization algorithms. These detailed results then assist in understanding the second set of results received from running these frameworks on simplified versions of four real-world HPC applications. Using this "symbiotic" evaluation approach, I contribute answers to the following research questions:

- What *systemic strengths and weaknesses* does each framework's optimization algorithm have?
- If any weaknesses are identified, how large is the *real-world impact* of said weaknesses?
- How high is the *time and effort* the programmer needs to expend to apply these frameworks to any given application?
- How can future frameworks *improve* their optimization algorithms and user experience?

The rest of this thesis is structured as follows. Chapter 2 provides the necessary background to understand the intricacies of the data placement optimization problem. Chapter 3 discusses related work, enumerating proposed data placement optimization frameworks and the evaluation conducted on them. Chapter 4 presents the three frameworks I evaluate from said related work in more detail. In Chapter 5, I evaluate and compare these frameworks using the aforementioned methodology. I discuss the obtained results in Chapter 6, where I also give recommendations for future placement optimization frameworks based on my results. This directly leads into the conclusion of this thesis in Chapter 7.

2. Background

As illustrated in the introduction, deploying heterogeneous memory systems seeks to serve diverse goals. While some of these goals are achievable rather simply (such as increasing memory capacity by installing lower-cost, slower memory), others are more difficult to satisfy. Into this second category fall the goals of increasing energy efficiency and increasing application performance. As I will explain in this chapter, this is because the performance and energy efficiency of the memory technologies used in heterogeneous memory systems not only depend on the technology itself but also on the application's individual *memory access behaviors*. Thus, optimizing the data placement in such systems requires (a) understanding this interaction between hardware characteristics and software behavior and (b) knowing how to identify the relevant access behaviors in an application.

To that end, this chapter provides an overview of the most commonly used memory types along with their characteristics (Section 2.1) and a review of the most prevalent approaches to obtain information about an application's memory-related behavior (Section 2.2). Also, this chapter demonstrates how to control the data placement in heterogeneous memory systems in practice (Section 2.3).

2.1. Memory Hardware

To meet the increasing performance and energy efficiency demands, hardware vendors have developed new *memory types*[1] to exist alongside the traditional main memory type, *dynamic random access memory* (DRAM). These new types, such as *high-bandwidth memory* (HBM) and *non-volatile memory* (NVM), are each optimized for one specific goal. For example, HBM is optimized for higher bandwidths and thus performance, while NVM is optimized for delivering higher capacities at lower energy consumption [32].

Focusing on the optimization of one characteristic at a time is necessary, as constructing one "best" memory type that combines the highest achievable performance, energy efficiency, and capacity is infeasible [41, 27]. Thus, systems must use multiple memory types together to meet the diverse memory-related demands of their users. Currently, most such heterogeneous memory systems use two memory types in tandem, typically DRAM plus either NVM or HBM. Hence, I am going to focus my discussion on these memory types.

[1] In this thesis, I use the term *memory type* to describe distinct hardware-level memory implementations, following the nomenclature of, e.g., Narayan et al. [41]. Other authors, such as Klinkenberg et al. [32] or Dulloor et al. [16] use the term *memory technology* for this.

Memory type	**Latency** (idle − loaded)	**Bandwidth** (peak)	**Power draw** (peak)
DRAM	80 − 230 ns	78 − 105 GB	55 − 60 W
NVM	169 − 800 ns	12 − 38 GB	5 − 15 W
HBM	130 − 194 ns	340 − 550 GB	45 − 60[a] W

[a] To the best of my knowledge, no direct power measurements are available for HBM as it is often located *on-chip*, complicating such measurements [3, 20]. These power figures are extrapolated from the energy efficiency results of Allen et al. [3] assuming a 55 − 60W window for DRAM.

Table 2.1.: General performance metrics for DRAM, NVM & HBM (data taken from [43, 3, 24, 45, 63, 39])

Generally, DRAM has traditionally provided good performance, i.e., high bandwidths and low latencies, at high capacities. However, due to reaching physical limitations, DRAM is unable to meet the rapidly increasing performance and capacity demands [44, 32]. Further, its high energy consumption has come under scrutiny in recent times. NVM provides higher capacities than DRAM with lower costs and energy consumption, at the cost of significantly lower bandwidths and higher latencies. Its performance and capacity place NVM between traditional main memory and secondary storage devices such as SSDs [16, 44, 63]. HBM provides significantly higher sustainable bandwidths than DRAM (circa 3× to 5× higher). Yet, it sacrifices latency and (at least in products available today) significant capacity in comparison to DRAM [47, 51, 39].

Table 2.1 demonstrates these general trends, showing typical ranges for the previously discussed key metrics for each memory type. However, it is wrong to assume that the concrete performance of any memory type falls randomly within the presented ranges. Rather, the achieved metrics depend on how the memory is accessed. Previous research identified the three following factors as most influential for memory performance [43, 24, 45, 63, 39]:

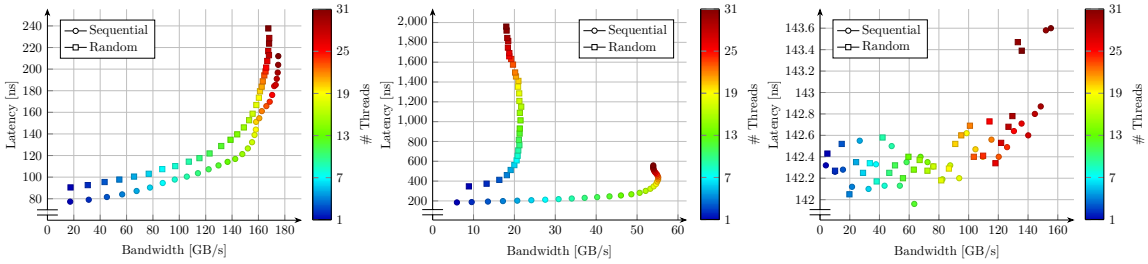
1. *Read-write ratio* of the memory accesses
2. The *pattern* of memory accesses (e.g., sequential vs. random accesses)
3. *Thread-level contention* (i.e., the number of parallel accessing threads)

Due to its importance for this thesis, I am going to briefly discuss how these three factors affect each memory type's performance.

2.1.1. Dynamic Random Access Memory

DRAM's performance is relatively stable in that neither different access patterns nor the accesses' read-write ratio affect its performance significantly. Regarding access patterns, DRAM's latency is only 10 % to 20 % higher and its bandwidth at most 10 % lower for random accesses than for sequential ones [32, 45, 63]. Further, power draw sees virtually no change between different access patterns [3]. Similarly, DRAM is minimally impacted by the read-write ratio of the accesses, i.e., how many writes are done for each read. At most, bandwidth decreases and power draw increases by 10 % each when increasing the number of writes in the workload [45, 32].[2]

[2]The impact of writes on DRAM *latency* is, to the best of my knowledge, not well explored. I discuss the reason behind this lack of latency information for writes in Section 2.2.



(a) Bandwidth-latency curve for DRAM

(b) Bandwidth-latency curve for NVM

(c) Bandwidth-latency curve for HBM

Figure 2.1.: Memory performance under different access patterns (plots taken from [32], measured with Intel's Memory Latency Checker [11])

Figure 2.1 (a) visualizes DRAM's performance in a *bandwidth-latency curve*. Such curves plot the changes in bandwidth (x-axis) and latency (y-axis) as the number of threads accessing the memory increases (indicated by color). The two sets of measurements for a sequential and random access pattern are only slightly separated, indicating DRAM's invariance to access patterns.

However, Figure 2.1 (a) also shows that DRAM is, in fact, significantly affected by thread-level contention. This pictures the differences between *idle latencies* and *loaded latencies* also seen in Table 2.1. Idle latencies are the latencies measured when no other accesses to the memory are done in parallel. In contrast, loaded latencies refer to the latencies measured when there are parallel and thus "competing" accesses. Such differences between idle and loaded latencies must exist for all memory types due to *Little's law*, which establishes a relationship between memory latency and bandwidth [60, 39]:

$$\text{Concurrency} = \text{Bandwidth} \times \text{Latency} \tag{2.1}$$

In Equation 2.1, *concurrency* refers to the number of memory request buffers being used [39]. Since it is impossible to increase this concurrency ad infinitum (as the number of available request buffers is determined by the hardware), an increase in bandwidth will inevitably cause an increase in latency at some point. Figure 2.1 (a) shows this consequence of Little's law for DRAM: As DRAM hits its concurrency limit due to being accessed by many threads in parallel, the curve's incline increases, indicating a sharper increase in latency for less gain in bandwidth.

2.1.2. Non-Volatile Memory

In contrast to DRAM, NVM is very sensitive to changes in access pattern, read-write ratio, and thread-level contention. Regarding access patterns, idle latencies almost double when switching from sequential to random reads (169ns vs. 309ns) [63]. As illustrated in Figure 2.1 (b), this already significant gap widens vastly for loaded latencies. Similarly, bandwidth deteriorates up to 75 % when switching from sequential to random accesses [24, 32]. Interestingly, power draw *de*creases by as much as 67 % for random over sequential accesses [48].

As for the read-write ratio, NVM's write performance is substantially lower than its read performance. This causes bandwidth to decrease from 38 GB/s to 12 GB/s when switching from a read-only to a read-write workload [45, 46]. At the same time, power draw surges, increasing up to 11× over read-only accesses [48]. Thread-level contention also affects NVM considerably more than DRAM. As Figure 2.1 (b) shows, NVM's bandwidth decreases by as much as 15 % from its peak when increasing the thread-level contention beyond NVM's physical concurrency limit. Latency increases even more severely, with, e.g., sequential read latencies rising from the forenamed 169ns to 600ns.

The reason for the access behaviors' large performance impact is NVM's physical implementation. Most NVM modules, such as the Intel Optane PMem module used by all previously cited works, employ so-called *phase-change memory* (PCM) [12, 44]. This explains NVM's lower write performance, since writing to PCM involves changing the resistance of a phase-change material through a thermal process. Said process takes longer and requires more energy than the purely electrical writes to DRAM. The cause for NVM's access pattern sensitivity is an implementation detail of the used Optane modules. They employ small DRAM caches to hide the higher access latencies of PCM [45, 63]. Such small caches and their prefetchers are highly sensitive to access regularity. Further, to better utilize these caches, Optane uses *access-combining read and write pending queues* [46, 63]. These combine physically adjacent read and write operations to increase access locality. However, they become a bottleneck when increasing thread-level contention, thus explaining NVM's sensitivity to this factor.

2.1.3. High-Bandwidth Memory

HBM's performance behavior is similar to DRAM in that it is mostly invariant to differing access patterns and read-write ratios. Yet, in contrast to DRAM, its performance is also unaffected by thread-level contention. Figure 2.1 (c) visualizes this fact: latencies stay almost constant while bandwidth monotonically increases with increasing thread-level parallelism. This is because HBM is typically implemented through specialized 3D-stacked DRAM modules (so-called *multi-channel DRAM*) [43, 47, 39], thereby "inheriting" DRAM's invariances while increasing tolerance to thread-level contention.

However, some intricacies regarding HBM's performance remain. While its latency remains unchanged when altering access behaviors, its idle latency is 65 % higher than DRAM [32]. Per Little's law, this higher latency implies a lower peak bandwidth than DRAM for random accesses. That is because the hardware prefetchers cannot make as many concurrent memory requests in these scenarios due to random accesses being unpredictable [47]. Also due to Little's law, utilizing HBM's higher peak bandwidth requires fully saturating its available memory-level concurrency. Applications may have difficulties in doing so, as Figure 2.1 (c) demonstrates: 31 threads do not supply enough memory-level parallelism to surpass DRAM in bandwidth. In such cases, HBM's bandwidth advantage may only be exploited by

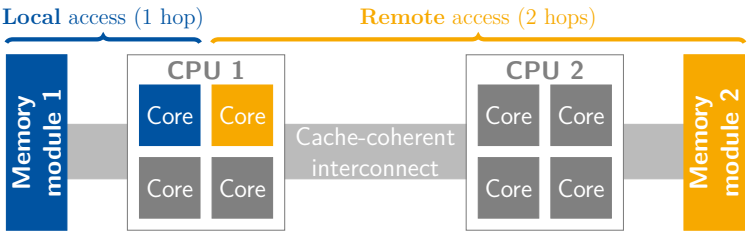


Figure 2.2.: Non-uniform memory access (NUMA) architecture (adapted from [64])

using *hyperthreading*, i.e., using more application threads than CPU cores [47, 3].

Concerning HBM's write performance, it is noteworthy that its achievable bandwidth improves when switching from a read-only to a mixed read-write workload. Klinkenberg et al. [32] report that a read-write workload achieves 38 % higher bandwidth than pure reads (440GB/s vs. 320GB/s). This behavior is diametrically opposite to NVM and further hints at the difficulties in saturating HBM's available concurrency.

2.1.4. Other Factors Affecting Memory Performance

The three discussed access behaviors are only a very small subset of performance-influential factors. For example, the *non-uniform memory access* (NUMA) architecture of modern systems also majorly affects memory performance. In NUMA architectures, some memory modules are physically closer to (and thus, faster to access for) one set of CPU cores than others [64, 21]. As depicted in Figure 2.2, the closer accesses are often referred to as *local*, while the other accesses are considered *remote*. Such remote accesses impact the performance of different memory types to different degrees. For example, NVM's performance declines significantly more than DRAM's when switching from local to remote accesses [45, 63].

Other performance-influential factors are the use of *non-temporal stores*, i.e., writes that bypass the cache hierarchy, or data alignment. Non-temporal stores can improve the write bandwidth for all memory types significantly (as much as 40 %), but are highly dependent on circumstance [3, 63]. Data alignment affects HBM's achievable bandwidth by up to 100 %, but does not impact DRAM [39].

And even this list is non-exhaustive (the interested reader may find more factors in the cited works). Yet, they further illustrate the main point of this section: The performance of different memory types is subject to manifold factors, some of which are more predictable and/or influencable, whereas others are less so.

2.2. Measuring Memory Access Behaviors

As presented in the previous section, plenty of factors affect memory performance. However, knowledge of these factors alone is insufficient to optimize the data placement in heterogeneous memory systems. This is because application programmers usually lack the knowledge required to manually identify the performance-relevant

factors discussed in Section 2.1 in such detail as to allow for data placement optimization [32]. Hence, optimizing data placement requires sound methods to determine a program's memory-related behavior, which I briefly present in this section.

2.2.1. Performance-Related Measurements

For performance-related program analysis, the following techniques exist [34, 18]:

1. *Static analysis*
2. *Dynamic profiling*, which can be further distinguished into
   a) Binary *instrumentation*
   b) Hardware-assisted *sampling*

Static analysis refers to analyzing a program's source code to obtain information about the program. Regarding memory-related behavior, static analysis can determine the read-write ratio and some (!) access patterns at a per-variable granularity [30]. Requiring no execution of the application, static analysis entails little overhead compared to the other measurement techniques.

However, its static nature imposes inherent theoretical limits on its capabilities. For example, *pointer aliasing* may break the $1:1$ correspondence between variables and distinct *data objects*. Pointer aliasing is a property known to be undecidable [50], which may cause static analysis to misattribute memory-related behavior. Further, some performance-relevant factors arise only during program execution (such as NUMA effects) or are otherwise undetectable without program execution [16, 34]. Thus, static analysis is best suited as a tool supplementing the information gained through dynamic profiling techniques.

Dynamic profiling (**profiling** for short) describes the act of executing a program while measuring certain aspects of it [25]. It is often used to, e.g., find the time spent in a certain function over the program's execution. However, it can also measure memory-related metrics such as the number of memory accesses or even how many cache misses occurred during program execution. These detailed metrics make profiling more versatile than static analysis for determining memory-related behavior; they can help compute the read-write ratio, access pattern, and thread-level contention of each memory object allocated by the program [16, 53, 27].

Within dynamic profiling, instrumentation and sampling differ in how they obtain the forenamed metrics. **Instrumentation** takes the approach of tracking each individual relevant CPU instruction in the final executable [53]. Tools employing instrumentation-based profiling, such as Intel's Pin [14, 37] or LLVM's AddressSanitizer [52], have the advantage of yielding high-accuracy data due to profiling each and every memory access. Thereby, instrumentation can enable a far more detailed analysis of memory access behaviors than static analysis.

However, these tools have a very high overhead. They not only require executing the program but also slow down the program's execution drastically. For instance, Intel's Pin introduces a $40\times$ slowdown in program execution time, making this pro-

filing method borderline unusable for production-scale applications [16, 61]. Further, instrumenting a program requires recompiling the program and linking it against the profiler [52, 53, 32]. The inserted instrumentation instructions may also prevent the compiler from performing optimizations, thus altering the application's behavior from the non-profiled version. Hence, instrumentation tools are unable to provide accurate hardware-level information such as the program's memory access latencies.

To address these problems of instrumentation, **sampling** does not profile each and every relevant instruction. Instead, it uses specialized hardware counters provided by modern CPUs (called *Processor Event-Based Sampling* (PEBS) for Intel [10, Vol. 3, Ch. 21] and *Instruction-Based Sampling* (IBS) for AMD [15]) to only periodically capture the desired profiling data.

As sampling only relies on these hardware counters, sampling-based profilers do not require recompiling the profiled application. Instead, they may be directly applied to the production-grade executable via the `LD_PRELOAD` mechanism of the Linux dynamic loader. Further, as they do not capture every CPU instruction, overheads are substantially lower than for instrumentation; typical sampling-based profilers such as Extrae [8, 27] and NumaMMA [57, 33] induce overheads between 1 % and 25 %, depending on the chosen sampling frequency. Due to being intertwined with the hardware, sampling can also provide more hardware-level information than instrumentation, such as accurate access latencies or cache hit rates.

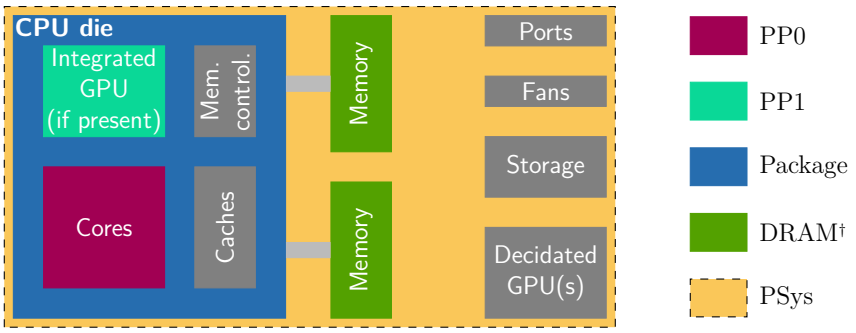
However, sampling has reduced accuracy compared to instrumentation since only a subset of instructions is profiled [16, 53]. Thus, sampling is unable to, e.g., identify more complex access patterns. Additionally, the metrics available to sampling-based profiling are processor-specific, with memory-related information being limited in some systems [27, 32]. For example, modern processors provide neither the latencies of nor detailed cache hit information for writes, which is due to the complexity of how writes to memory are effectuated in modern architectures.

2.2.2. Energy Measurements

Energy consumption can be measured via two different approaches: *external instrumentation* or *intra-hardware measurements*. **External instrumentation** involves attaching external power meters to the system or a selection of its components. While straightforward in its approach, it poses practical difficulties due to requiring specialized equipment (especially when taking measurements for only certain system components such as the memory modules) and physical deployment [31, 4].

Because of these difficulties, modern Intel and AMD processors include hardware registers that enable **intra-hardware measurements** without external tooling. This *Running Average Power Limit* (RAPL) interface enables energy monitoring and regulation of different combinations of components within the system (so-called *domains*) [10, Vol. 3B, Ch. 16, 49]. Figure 2.3 shows the different domains and their names for Intel processors;[3] however, not all domains are available for all processors.

[3] In the following, I focus my discussion on Intel systems, since I conduct my tests later in this thesis on an Intel system.



†Despite the name, this domain includes *all* memory types attached to the CPU

Figure 2.3.: RAPL domains for Intel processors (adapted from [31, 49])

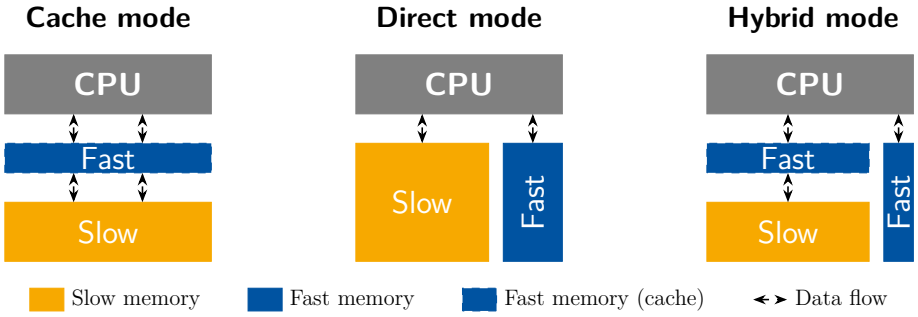


Figure 2.4.: HMS configurations with two memory types (adapted from [45])

Tools such as LIKWID [59] or Linux' `perf` may then periodically access RAPL's hardware registers and capture the energy data, introducing negligible (< 1 %) overhead [31]. This data linearly correlates with externally made measurements (correlation coefficient ≥ 0.99), but only at a constant offset [31, 4, 49]. Thus, RAPL's energy figures may not reflect reality exactly in all circumstances, meaning it cannot serve as a full replacement for external instrumentation. However, due to the high linear correlation, RAPL still provides an internally consistent frame of reference to evaluate how changes to an application generally affect its energy consumption.

2.3. Controlling Data Placement in an HMS

Once the programmer has used the knowledge from Sections 2.1 and 2.2 to optimize their application's data placement, the programmer must then facilitate these optimized placements. How exactly they can do so depends on the configuration of the heterogeneous memory systems. Heterogeneous memory systems with two memory types (which I focus on in this thesis) can be configured in three ways: *cache mode*, *direct mode*, or *hybrid mode*.[4] Figure 2.4 depicts these configuration options and their differences. As the figure shows, hybrid mode is a mere combination of cache

[4]The names of the modes differ from one hardware vendor to another. I chose the given names to be brand-neutral.

and direct mode, which is why I will not discuss it further.

Cache mode allocates the "faster" memory type in the HMS (e.g., DRAM in a DRAM + NVM system) as an additional cache layer in front of the other, "slower" memory type [45]. Therefore, the hardware and operating system control the application's data placement; the programmer need not (and, in fact, cannot) perform any placement optimization themselves.

Cache mode thereby lets the programmer use the system's memory heterogeneity without altering their application's code. Yet, this ease of use sacrifices significant performance over manual placement optimization [47, 53, 3, 45]. In some circumstances, it may even deteriorate memory performance below the level of the slower memory type in the system. Further, cache mode nullifies the potential energy efficiency benefits of heterogeneous memory systems due to running all memory types in the system under constant load. Also, cache mode makes significant portions of the system's full memory capacity (i.e., the sum of the capacities of the memory types) unusable since the faster memory is used as a cache layer.

Thus, the programmer must manually optimize their application's data placement using **direct mode** to fully exploit the advantages of heterogeneous memory systems. Direct mode exposes all of the system's memory types to the user level, typically as separate NUMA domains or block devices [47, 45].

The programmer has multiple choices to control their application's data placement in direct mode. If they only desire coarse-grained control, they may use NUMA control tools such as `numactl` to place *all* application data into a specific memory type. Yet, this does not exploit the system's memory heterogeneity in any way, as it does take the performance differences discussed in Section 2.1 into account. Thus, to maximize their usage of heterogeneous memory, the programmer ought to control their application's data placement for each individual data object [41, 45].

To achieve such fine-granular control, the programmer needs to either modify their application's source code or use *interception tools*. For the former option, special allocation libraries such as `memkind` [7] provide drop-in replacements for the `malloc` family of memory allocation routines. Yet, directly modifying the source code imposes a high burden for the programmer and intertwines the placement optimization with the rest of the source code [53, 27, 33]. Further, the programmer cannot (easily) modify the internal allocation calls made by standard library data structures such as C++'s `vector` class.

Interception tools such as FlexMalloc [23] mitigate these drawbacks, using only configuration files instead of source code modification. Listing 1 shows how to use these configuration files to direct data placement. The first configuration file (Listing 1 (c)) is system-specific and lists the available memory allocators. These allocators direct the made allocations to the different memory types. In this example, the system contains DRAM (allocated to through the default `posix` allocator) and NVM (allocated to through the `pmem` allocator from the `memkind` library). The former has a size of 2048MB, while the latter is exposed as a block device and self-reports its capacity to FlexMalloc. Note that the usage of the `memkind/pmem` allocator still requires installation of the `memkind` library. The second configuration

```
1  int main(void) {
2      double *a = (double *) malloc(...);
3      std::vector<double> b;
4
5      // Computation involving a & b
6
7      free(a);
8  }
```

(a) Example C++ program (file named `compute.cpp`)

```
compute.cpp:2 @ posix
compute.cpp:3 @ memkind/pmem
```

(b) FlexMalloc location file

```
# Memory configuration for allocator posix
Size 2048 MBytes
# Memory configuration for allocator memkind/pmem
@ /mnt/pmem0 @ /mnt/pmem1
```

(c) FlexMalloc configuration file

Listing 1: Using FlexMalloc to control application data placement

file (Listing 1 (b)) is application-specific, listing for each allocation call which allocator (and thus, memory type) to use. In this example, array `a` (allocated in line 2 of `compute.cpp`) is placed into DRAM via the `posix` allocator and vector `b` (allocated in line 3) is placed into NVM via the `memkind/pmem` allocator.

While interception tools mitigate some disadvantages of manual source code modification, they do not alleviate all issues. They still require the programmer to go through the application's code and decide the placement of each allocated object [61, 27, 32].[5] Further, both approaches do not offer portability; they require "hardcoding" the specific allocators and memory types into them instead of providing a portable mechanism to detect the memory types present in a system at runtime.

Additionally, FlexMalloc distinguishes allocations only via their *callstack*, i.e., all function calls leading up to the allocation call. As such, FlexMalloc cannot separate objects that are allocated within a loop, since they all have the same allocation callstack. Yet, manual source code modification is also unable to deal with such loop allocations unless the programmer introduces branches with different allocation calls into the loop; at that point, FlexMalloc would also be able to distinguish the loop allocations since they now have different callstacks.

[5] FlexMalloc provides a fallback allocator mechanism through which the programmer may specify where to allocate objects not listed in the location file. Yet, this still requires the programmer to consciously *decide* for each object whether to omit it from the configuration file.

3. Related Work

Numerous works have proposed techniques for optimizing data placement in heterogeneous memory systems. Efforts therein range from simple heuristics to advanced, automated frameworks. The hardware studies cited throughout Section 2.1 offer first heuristics to guide usage of the different memory types based on their performance characteristics. For instance, Peng, Gokhale, and Green [45] recommend placing write-intensive objects into DRAM in DRAM + NVM systems because of NVM's low write performance. Yang et al. [63] further recommend placing objects with random accesses into DRAM, as they identified this access pattern to be detrimental for NVM. Similar recommendations exist for HBM + DRAM systems: Peng et al. [47] and Allen et al. [3] advise placing randomly accessed objects into DRAM rather than HBM due to HBM's higher latency.

The forenamed studies thoroughly tested these heuristics for their efficacy, finding them to significantly improve performance over cache mode setups. Peng, Gokhale, and Green even evaluated the energy efficiency implications of their write-aware placement, showing that it considerably reduces energy consumption. However, in their evaluation, the cited works did not discuss the difficulty of applying these heuristics to real-world applications. However, this is unsurprising, given that the focus of the aforementioned works lies in the evaluation of the different memory types rather than providing a user-focused placement optimization workflow.

Such optimization workflows (which I will refer to as *frameworks* for the rest of this thesis) exist aplenty, using application profiling to automatically identify performance-critical data objects. However, they differ in the extent to which they implement and build upon the aforementioned heuristics. For example, Servat et al. [53] use a simple cache miss density metric to guide data placement, a metric not found in any of the named studies. In their later work [27], they amend this metric with a bandwidth-aware object classification. With that, they seek to reduce NVM access contention in DRAM + NVM systems, which the hardware studies did find to be performance-relevant. Klinkenberg et al. [33] use an analytical model to predict how much time the application would spend accessing each object if it were placed in a specific memory type. Said model differentiates reads and writes, but not other performance-influential access behaviors such as access patterns. Dulloor et al. [16] take a very similar approach of estimating access times for their placement optimization but focus on differentiating access patterns instead of reads and writes. To that end, they employ instrumentation-based profiling rather than the sampling used by all previously listed works. Narayan et al. [41] experiment with entirely different profiling metrics not used in any other work. They classify objects into latency- and bandwidth-sensitive using the memory's reorder buffer (ROB) stall time. Wen

et al. [61] do not use any heuristics or analytical models for their placement optimization. Instead, they use a "differential analysis" [61] approach, in which they profile the application multiple times with different system configurations to detect performance-relevant objects.

Other frameworks focus on specific types of applications, seeking to use domain knowledge to achieve better placement decisions. Lasch et al. [35] target in-memory databases, building their placement optimization around what database accesses are favorable for each memory type. Han et al. [21] optimize data placement for deep learning model training in HBM + DRAM systems. To that end, they define custom notions of "tensor hotness" and "tensor lifetime" for deciding what tensors should be placed into HBM to accelerate the training process. Wu, Ren, and Li [62] target task-based applications, introducing the concept of "representative tasks" to minimize profiling overhead. Also, they use a machine-learning model for their placement decisions, differing from all forenamed works.

Still other frameworks implement a multi-goal placement optimization instead of targeting application performance alone. Gupta et al. [19] design heuristics for HBM + DRAM systems to optimize the application's data placement for both improved performance and increased HBM reliability. Katsaragakis et al. [29] employ Pareto optimization to conjointly optimize for performance, energy efficiency, and memory wear reduction in DRAM + NVM systems.

As evident by this list, previous work does not lack ingenuity in terms of possible optimization approaches. Yet, as already mentioned in the introduction to this thesis, said work lacks a comprehensive evaluation of these different approaches. Virtually all cited works test their frameworks only on an arbitrary selection of *proxy applications*, i.e., simplified versions of production-level HPC applications. These applications do not cover the full range of *all* performance-influential access behaviors discussed in Section 2.1, especially not in a systematic manner allowing for an *in-depth* evaluation of each approach's efficacy. Further and especially, the cited works rarely compare their newly developed frameworks to one another; of the cited works, only Wu, Ren, and Li [62] and Jordà et al. [27] make such comparisons. Yet, such comparisons in particular are a strict requirement to reason about (a) what benefits and drawbacks each optimization approach has and (b) how to potentially combine the advantages of the different frameworks.

Such comparisons should also include the frameworks' user experience because the user experience heavily influences how much use the frameworks see in future heterogeneous memory systems. However, the previously cited works cover at most how many application code changes each framework requires, despite "user experience" encompassing more than just code changes (e.g., the time the framework takes for its placement optimization or the effort required to use the framework). To address these shortcomings, the following chapters evaluate and compare three of the frameworks cited in this chapter with regard to both their placement optimization algorithms and their user experience. This will provide future research with the knowledge as to what optimization approaches should and should not be pursued further, along with recommendations for improving the user experience.

4. Tools for Automated Data Placement Optimization

As noted in Section 2.3, manually optimizing the data placement in heterogeneous memory systems is tedious and difficult to do well for the programmer. This sparked the creation of the several frameworks presented in Chapter 3, which, as noted therein, must be evaluated in-depth to guide future work on HMS data placement optimization. Yet, the list presented in Chapter 3 is so extensive that it is infeasible for a singular thesis to evaluate *all* named frameworks in-depth. Hence, I focus my evaluation on three representative frameworks that cover a wide range of optimization approaches: *H2M* of Klinkenberg et al. [32, 33], *ecoHMEM* of Jordà et al. [53, 27], and *ProfDP* of Wen et al. [61].

However, drawing conclusions from the evaluation results in Chapter 5 requires understanding the mechanisms underlying each framework. To that end, this chapter presents the placement optimization algorithms each framework uses for its placement decisions. Also, I discuss the overarching workflow these algorithms are embedded in, focusing on the "intended workflow" each framework seeks to provide; any practical issues or bugs these frameworks entail are instead discussed in Chapter 5.

4.1. H2M

H2M started as a partially manual placement optimization library; for each object, the programmer specified how that object is accessed, which H2M used as the basis for its optimization. Only later did its authors evolve it into a fully automated framework. Figure 4.1 shows H2M's general workflow. It profiles the application once, obtaining memory access metrics for each object dynamically allocated by the program [33]. Together with performance data for the memory types present in the system, H2M uses this information to optimize the application's data placement. It then executes the optimized placements with the FlexMalloc interception tool.

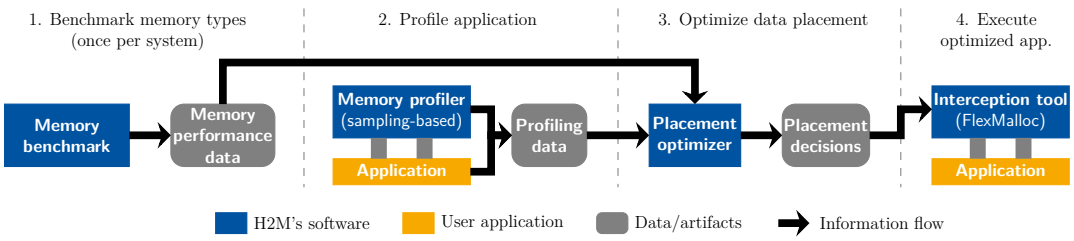


Figure 4.1.: H2M's workflow (adapted from [33])

4.1.1. Optimization Algorithm

H2M bases its placement optimization on access time estimates. For each of the application's objects, it estimates how much time the application would spend accessing that object if it were placed in a certain memory type. H2M subsequently seeks to minimize this access time via its placement decisions. For this estimation, H2M employs a sophisticated analytical model that uses the profiling information obtained in the first steps of the workflow. This model specifically is why I chose H2M as one of the frameworks I evaluate, as it represents several other frameworks using similarly sophisticated models for their optimization.

$$\mathrm{AT}_i^{\mathrm{acc,mem}} = \frac{N_i^{\mathrm{acc}} \cdot \mathrm{Lat}_{\mathrm{mem}}}{\mathrm{CLS} \div \texttt{sizeof(double)}} \cdot (1 - \mathrm{CHR}_i^{\mathrm{acc}}) + \frac{N_i^{\mathrm{acc}} \cdot \mathrm{CLS}}{\mathrm{BW}_{\mathrm{mem}}^{\mathrm{acc}}} \cdot \mathrm{CHR}_i^{\mathrm{acc}} \quad (4.1)$$

Equation 4.1 shows the forenamed model estimating the access time for object i if placed in memory type mem; acc $\in \{\mathrm{read}, \mathrm{write}\}$ determines whether to estimate the access time for reads or writes. N_i^{acc} denotes the number of profiled (read or write) accesses to i, and $\mathrm{CHR}_i^{\mathrm{acc}} \in [0, 1]$ is the cache hit rate of these accesses. For reads, $\mathrm{CHR}_i^{\mathrm{acc}}$ is the *last level cache* (LLC) hit rate; for writes, it is the *level 1 data cache* (L1D) hit rate since other cache data is unavailable for writes as discussed in Section 2.2 [33]. CLS = 64B is the cache line size of the system. Its usage reflects that memory accesses gather one cache line of data at a time rather than singular bytes. $\mathrm{BW}_{\mathrm{mem}}^{\mathrm{acc}}$ and $\mathrm{Lat}_{\mathrm{mem}}$ denote the bandwidth and latency of the memory type, with the bandwidth further depending on the access type.

Put briefly, Equation 4.1 consists of two parts. The first summand determines the access time for all accesses that miss the cache and thus incur the memory's full latency. The second summand determines the access time for the accesses that hit the cache and are thus bound by the memory's bandwidth instead of its latency.

From $\mathrm{AT}_i^{\mathrm{acc,mem}}$, H2M computes the total access time for each object as the sum of its read and write access times, i.e., $\mathrm{TAT}_i^{\mathrm{mem}} = \mathrm{AT}_i^{\mathrm{read,mem}} + \mathrm{AT}_i^{\mathrm{write,mem}}$. It then uses $\mathrm{TAT}_i^{\mathrm{mem}}$ to assign each object a *value* in a *0/1 knapsack problem* that represents the placement optimization problem [33]. The knapsack's capacity is the capacity of the faster memory type in the system,[1] with each object's weight being its size. The value of each object is the access time saved by placing that object into the faster memory type instead of the slower:

$$\mathrm{V}_i = (\mathrm{TAT}_i^{\mathrm{slow}} - \mathrm{TAT}_i^{\mathrm{fast}}) \cdot \mathrm{Freq}_S \quad (4.2)$$

Scaling the value with the sampling frequency Freq_S offsets the underestimation of the access time incurred by only sampling a subset of memory accesses [33].

H2M's optimization uses not one fixed 0/1 knapsack formulation but offers the user a choice between three different formulations [33]:

[1] H2M mainly assumes a system with one faster and one slower memory type. For systems with more than two memory types, H2M iteratively solves the knapsack: In the first iteration, the knapsack represents the fastest memory type, in the second iteration the second-fastest, etc. [33]

- *Initial data placement* (IDP)
- Initial data placement *with lifetimes* (IDP-LT)
- *Phase-based data placement* (PBDP)

In **IDP**, H2M assumes that each object is allocated for the entire runtime of the application. This allows H2M to formalize the decision problem as a basic 0/1 knapsack, represented by the following integer linear program [33]:

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{n} x_i \cdot \mathrm{V}_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_i \cdot \mathrm{Size}_i \leq \mathrm{FastMemCap} \qquad x_i \in \{0,1\} \end{aligned} \tag{4.3}$$

Object i is placed into fast memory if and only if $x_i = 1$ in the optimal solution of Equation 4.3. Yet, the assumption that all objects are "alive" for the application's entire runtime is often false, leading to suboptimal usage of the available fast memory space [33]. Thus, **IDP-LT** additionally considers each object's *lifetime*, i.e., its allocation and deallocation times, allowing H2M to potentially place more objects into fast memory. To do so, IDP-LT solves Equation 4.3 recursively for objects with overlapping lifetimes [33].

PBDP allows H2M to migrate objects between memory types at runtime to further optimize the usage of the limited fast memory space. For that, the programmer divides their application into phases (explained in more detail in Section 4.1.2). In its optimization, H2M can then decide to migrate objects when transitioning from one phase into the next [33]. Of the three frameworks I cover, H2M is the only framework with such capabilities.

PBDP solves Equation 4.3 separately for each phase in the program. However, to take the overhead of runtime data migration into account, the objective function of Equation 4.3 changes to

$$\max_{x} \sum_{i=1}^{n} x_i \cdot \hat{\mathrm{V}}_i - a_i \left(y_i (1 - x_i) \mathrm{TC}_i^{\mathrm{fast} \to \mathrm{slow}} + (1 - y_i) x_i \mathrm{TC}_i^{\mathrm{slow} \to \mathrm{fast}} \right) \tag{4.4}$$

Migrating object i causes overhead only if it was allocated prior to the current phase, which the constant $a_i \in \{0,1\}$ indicates. Further, overheads only occur if the object is placed in different memory types in the prior and current phases. To determine this, the constant $y_i \in \{0,1\}$ indicates whether the object was placed in the faster ($y_i = 1$) or slower memory type ($y_i = 0$) in the previous phase, with x_i having the same semantics as before. $\mathrm{TC}_i^{\mathrm{slow} \to \mathrm{fast}}$ and $\mathrm{TC}_i^{\mathrm{fast} \to \mathrm{slow}}$ denote the overhead of transferring the object from slow into fast and from fast into slow memory, respectively. This transfer time equals the object's size divided by the copying bandwidth [33].

Also, $\hat{\mathrm{V}}_i$ differs from the previous object value V_i in that it only includes the profiled accesses starting from the current phase instead of all accesses across the entire program's runtime.

4.1.2. User Experience

For the end user, H2M aims to be a fully automated framework [33]. To investigate whether H2M meets this goal, I subsequently detail for each step in H2M's workflow whether (a) it provides all necessary tooling to automate this step and whether (b) it guides the user in the application of said tools.

Step 1 of H2M's workflow involves discovering what memory types are present in the target system and consequently benchmarking them (cf. Figure 4.1). In contrast to the other steps, this step only needs to be done once *per system* instead of once per application. For the discovery task, the programmer can use the `hwloc` tool [6], which Klinkenberg et al. have extended for this purpose. For the benchmarking task, H2M provides a benchmarking suite[2] comprising the STREAM [40] and lmbench [55] memory benchmarks and Intel's Memory Latency Checker [11]. The programmer only needs to set up the suite's execution environment (which H2M documents how to do), after which the suite automatically runs all tests necessary to determine the memory metrics required by the optimizer (i.e., $\mathrm{BW}_{\mathrm{mem}}^{\mathrm{acc}}, \mathrm{Lat}_{\mathrm{mem}}$, and the copy bandwidths for calculating $\mathrm{TC}_i^{\mathrm{slow}\to\mathrm{fast}}, \mathrm{TC}_i^{\mathrm{fast}\to\mathrm{slow}}$). The suite outputs these results into a configuration file that H2M can directly use.

Step 2 is the application profiling step, for which H2M uses the NumaMMA sampling profiler [57]. As mentioned in Section 2.2, the programmer need not recompile their application for the profiling. However, they also *must not* recompile their application *after* the profiling step has taken place; otherwise, the optimized placements cannot be executed correctly. This fact is, however, not documented, despite H2M otherwise providing sufficient documentation for the profiling step.

One exception where the programmer does need to modify their application (*before* the profiling step) is if they want to use PBDP. In this case, they must add calls to `h2m_phase_transition` in their application (to denote a transition between two application phases) and link it to the H2M runtime [33]. The programmer is given no guidance on how they should identify phases in their application or what granularity the phases should be for optimal results. Klinkenberg et al. only recommend using common tracing tools such as Intel's VTune [13] for this purpose.

In **step 3**, the programmer provides the placement optimizer with the memory metrics and profiling data from the first two steps. The optimizer then outputs its placement decisions into a JSON file [33]. For the optimizer to run, the programmer must also manually input the following information: (1) the capacity of the faster memory type in the system, (2) whether to use IDP, IDP-LT, or PBDP, and (3) how many threads were used to execute the application in the profiling step. Yet, H2M does not document that this information must be provided manually by the user, nor does it document how the user ought to provide it. This is especially consequential since (1) and (2) must be specified to the optimizer via opaque environment variables. Also, H2M gives the user no guidance on when using PBDP over the IDP variants might be beneficial for the application.

[2]The benchmarking suite, along with all other software that H2M requires, is available under `https://gitlab.inria.fr/h2m`.

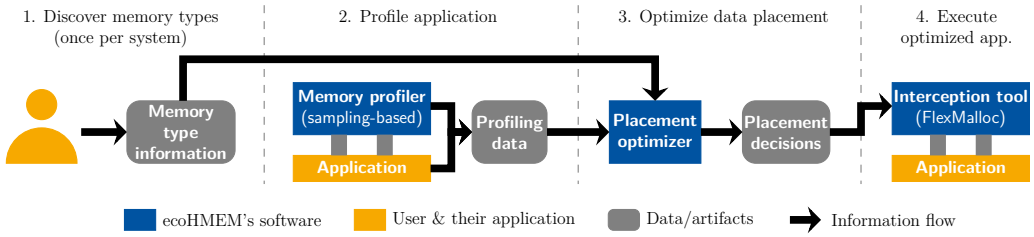


Figure 4.2.: ecoHMEM's workflow

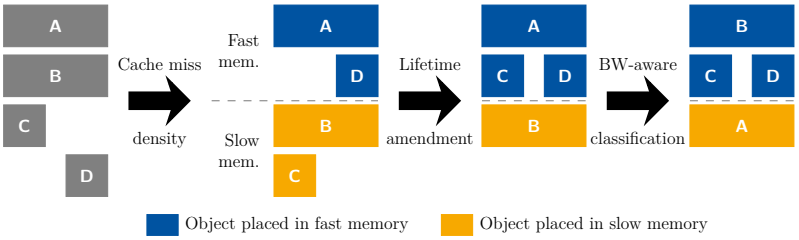


Figure 4.3.: ecoHMEM's three-step optimization algorithm

For **step 4**, the execution of the application with the optimized placements, H2M uses a customized FlexMalloc fork. Klinkenberg et al. have modified FlexMalloc to work with the JSON output from the optimizer, which, in contrast to the normal FlexMalloc presented in Section 2.3, can also distinguish allocations made by the same callstack. However, as with step 3, H2M does not document the made customizations nor that the JSON file needs to be input via an undocumented environment variable (instead of FlexMalloc's usual command-line interface).

4.2. ecoHMEM

ecoHMEM, in contrast to H2M, immediately started as a fully automated placement optimization framework for HBM + DRAM systems [53]. Its authors later expanded it with bandwidth-aware heuristics for DRAM + NVM systems [27]. In its workflow, it is very similar to H2M, as Figure 4.2 shows. The only difference between the two is in the first step, with ecoHMEM having the user manually provide basic information on the system's memory types instead of running a full memory benchmark.

4.2.1. Optimization Algorithm

ecoHMEM's placement optimization is a three-step process, which Figure 4.3 depicts. First, ecoHMEM assigns each dynamically allocated object a value via a simple cost metric, which it uses to derive initial placement decisions [27]. In a second step, ecoHMEM amends these placements using object lifetime information, thus placing more objects into fast memory. Lastly, ecoHMEM classifies all objects based on their bandwidth demand and alters their placements accordingly. This three-step approach is why I chose ecoHMEM as one of the frameworks I evaluate since many frameworks use such an iterative refinement approach.

In **step 1**, ecoHMEM uses a cache miss density metric to determine the value of placing object i into memory type mem [27]:

$$V_i^{\text{mem}} = \frac{N_i^{\text{readMiss}} \cdot c_{\text{mem+1}}^{\text{read}} + N_i^{\text{writeMiss}} \cdot c_{\text{mem+1}}^{\text{write}}}{\text{Size}_i} \tag{4.5}$$

N_i^{readMiss} and $N_i^{\text{writeMiss}}$ denote the number of read and write cache misses profiled for the accesses to object i. As with H2M, read cache misses are LLC misses, while write cache misses are L1D misses since other cache data is unavailable for writes [27]. $c_{\text{mem}+1}^{\text{read}}$ and $c_{\text{mem}+1}^{\text{write}}$ are user-definable coefficients representing the cost of each read and write miss for the memory type. Note that the formula uses the coefficients of the *next slower* memory type (denoted as mem + 1 in Equation 4.5) so that V_i^{mem} represents the *gain* of placing object i into memory type mem instead of mem + 1.

ecoHMEM uses this metric in a greedy relaxation of the 0/1 multiple knapsack problem to derive initial placements [27]. It sorts the memory types by their speed and the objects by their value in descending order. Subsequently, it fills the fastest memory type with as many high-value objects as possible before moving to fill the next slower memory. Of note for this optimization is that ecoHMEM, in contrast to H2M, distinguishes objects solely by their callstack [53]. This downside is caused by ecoHMEM's profiler, Extrae [8], which considers all allocations made through the same callstack to be one object. Further, Extrae does not accumulate the profiling metrics across all allocations *associated with* the same callstack. Instead, it reports only the maximum recorded for *one* allocation as that object's profiled metrics.

Step 2 modifies the obtained placements, using object lifetime information to fit more objects into faster memory spaces. Therein, ecoHMEM prioritizes objects with high values (as determined in step 1) to "move up" into faster memory. Interestingly, Jordà et al. do not mention this second step in their description of ecoHMEM, with me only finding it through source code inspection. Said inspection also reveals that step 2 is hardcoded for DRAM + NVM systems, while step 3 (which Jordà et al. specifically designed for DRAM + NVM systems [27]) is not.

Step 3 aims to minimize the bandwidth usage of the slower NVM (in DRAM + NVM systems) to avoid thread-level contention deteriorating NVM's performance [27]. For this step, Jordà et al. made the following observations when investigating application profiling data [27]: (1) Objects/callstacks with many associated allocations typically have a shorter lifetime, and (2) objects with shorter lifetimes usually have a uniform bandwidth demand over their lifetime (in contrast to long-living objects). To use these observations, ecoHMEM classifies all objects into three categories via their average bandwidth usage and number of associated allocations:

- *Fitting*: objects currently placed in DRAM with few associated allocations and low bandwidth usage
- *Streaming-D*: objects currently placed in DRAM with many associated allocations and low bandwidth usage
- *Thrashing*: objects currently placed in NVM with many associated allocations and high bandwidth usage

Following the aforementioned observations, objects classified in Streaming-D likely have low bandwidth demands over their entire lifetime. Hence, they can be reassigned into NVM instead of DRAM as they are not performance-critical [27]. Analogously, objects classified in Thrashing likely have high bandwidth demands over their lifetime. Hence, ecoHMEM places these objects into DRAM, reassigning objects from the Fitting category into NVM to free up DRAM space as necessary.

4.2.2. User Experience

ecoHMEM makes similar promises regarding its automation as H2M [27]. As such, I subject ecoHMEM to the same analysis as H2M, investigating whether it provides all tools for each workflow step and whether it guides the user through their usage.

Step 1 requires the programmer to identify all memory types present in their system and manually write them down in a configuration file, along with the $c_{\text{mem}}^{\text{read}}$ and $c_{\text{mem}}^{\text{write}}$ weighting coefficients. Yet, contrary to H2M, ecoHMEM does not guide the user on how they should identify their target system's memory types, nor how they should intelligently choose the weighting coefficients. Jordà et al. only mention that the coefficients should represent the read and write latencies of the memory types [27]. However, this assertion is inconsistent with the example configuration files they provide alongside ecoHMEM.[3] In said examples, they set $c_{\text{mem}}^{\text{read}} = 5, c_{\text{mem}}^{\text{write}} = 10$ for NVM and both coefficients to 1 for DRAM, which are not correct latency numbers in either the absolute or relative sense (cf. Section 2.1). Therefore, choosing these coefficients poses a hyperparameter optimization problem for the user.

Also, the documentation omits the crucial fact that the order in which the memory types appear in the configuration file is important. The optimizer presumes the memory system in the first line of the configuration file to be the fastest, the second line to list the second-fastest memory type, etc., regardless of the chosen coefficients.

For **step 2**, the profiling step, ecoHMEM uses the Extrae sampling profiler. As with H2M, the user must not modify their application after profiling has taken place, which ecoHMEM also does not document. Otherwise, ecoHMEM provides the user ample documentation on how to conduct the profiling step.

In **step 3**, ecoHMEM uses the profiling data along with the memory type configuration file to optimize the placements, yielding a standard FlexMalloc location file (cf. Section 2.3, Listing 1 (b)). In contrast to H2M, ecoHMEM sufficiently documents how to use this placement optimizer. Further, Jordà et al. provide shell scripts that assist the user in executing all steps of ecoHMEM's workflow (bar the initial discovery step). While H2M provides similar scripts, ecoHMEM actually documents those scripts and what the user needs to modify in them to fit their application. Thus, ecoHMEM is more ergonomic to use than H2M.

Step 4 uses FlexMalloc to execute the optimized placements. Since ecoHMEM does not use a customized FlexMalloc fork, using the optimizer's output requires the user to install a heterogeneous memory library such as `memkind`. Yet, Jordà et al. fail

[3] Both available under `https://github.com/accelcom-bsc/ecoHMEM`

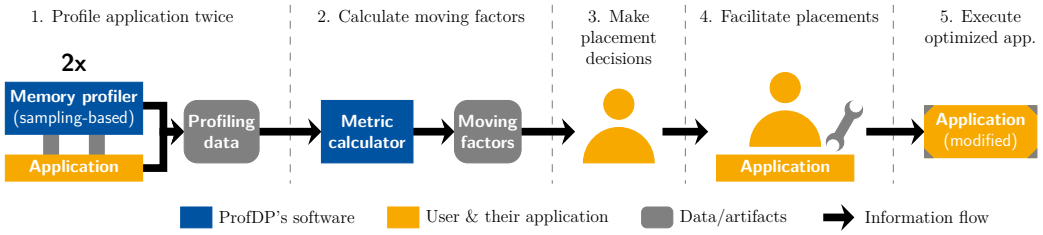
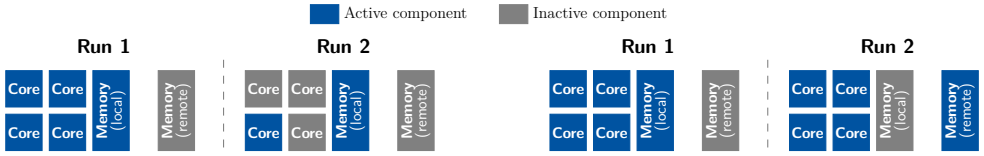


Figure 4.4.: ProfDP's workflow



(a) Bandwidth-aware approach (b) Latency-aware approach

Figure 4.5.: ProfDP's two differential analysis approaches (adapted from [61])

to document this dependency. Apart from this lapse, ecoHMEM sufficiently guides the user through the necessary FlexMalloc setup. For example, the documentation shows the user how to manually create the required FlexMalloc configuration file listing the system's memory types (cf. Section 2.3, Listing 1 (c)).

4.3. ProfDP

ProfDP is the framework of Wen et al. [61]. In contrast to the other presented frameworks, it is not a fully automated toolchain. Rather, it is a "lightweight profiler" [61] that merely assists the programmer in manually optimizing their application's data placement. Figure 4.4 shows this difference in workflow. Instead of making placement decisions itself, ProfDP only outputs so-called *moving factors*, leaving the usage of these metrics entirely up to the programmer.

4.3.1. Optimization Algorithm

ProfDP's optimization profiles the application twice with different system configurations [61]. It then uses the difference in *average access latencies* between these two runs to identify performance-critical objects, which should be placed in the fastest available memory. While this "differential analysis" [61] approach is unique to ProfDP, its usage of the average access latency represents multiple frameworks using similarly detailed metrics for their optimization. This is why I chose ProfDP as one of the frameworks I evaluate.

ProfDP offers two different approaches to the forenamed differential analysis [61], which Figure 4.5 shows. Its *bandwidth-aware* approach aims to identify the application's bandwidth-demanding objects, while the *latency-aware* approach seeks to find latency-sensitive objects. Wen et al.'s rationale behind having two approaches is that they assume an HMS to contain one faster and one slower memory type,

with the slower memory having *either* higher latency *or* lower bandwidth than the faster memory [61]. Thus, placement optimization should focus on placing either bandwidth- or latency-sensitive data objects into the faster memory.

In both approaches, the first profiling run is a reference run that uses the application's desired parallelism and places its data on a local NUMA domain. How the second run changes this configuration depends on the chosen optimization approach.

In the **bandwidth-aware** approach, the second run changes the application's parallelism to use only one singular core, as Figure 4.5 (a) shows. Wen et al. motivate this approach via Little's law, by which bandwidth should increase proportional to the application's parallelism unless latency increases [61]. Hence, they deem an object bandwidth-***in***sensitive "if the average latency of its memory accesses does not change when the program [...] scales from one [...] to more [...] cores" [61].

The **latency-aware** approach keeps the application's parallelism fixed across runs but places the application data on a remote NUMA domain in the second run, as Figure 4.5 (b) depicts. Wen et al. base this approach on the expectation that an object is latency-***in***sensitive "if the average latency of its accesses does not change when the program [...] runs on a machine with higher memory access latency" [61].

In both approaches, ProfDP profiles the average access latency for each object in each run (hereafter denoted as $\overline{\mathrm{L}}_i^{\mathrm{Run}x}$ for object i). Contrary to H2M and ecoHMEM, ProfDP samples this information not only for dynamically allocated objects but also globally allocated static data [61]. Using the latency data, ProfDP assigns each object a *bandwidth sensitivity* (BS_i) or a *latency sensitivity* (LS_i), depending on the chosen optimization approach. Mirroring Wen et al.'s expectations of when an object is bandwidth- or latency-(in)sensitive, these metrics are defined as the relative increase in average access latency between the two runs [61]:

$$\mathrm{LS}_i = \frac{\overline{\mathrm{L}}_i^{\mathrm{Run2}} - \overline{\mathrm{L}}_i^{\mathrm{Run1}}}{\overline{\mathrm{L}}_i^{\mathrm{Run1}}} \qquad \mathrm{BS}_i = \frac{\overline{\mathrm{L}}_i^{\mathrm{Run1}} - \overline{\mathrm{L}}_i^{\mathrm{Run2}}}{\overline{\mathrm{L}}_i^{\mathrm{Run2}}} \tag{4.6}$$

ProfDP weighs the obtained sensitivity (denoted as S_i below) with the object's relative size in comparison to the application's memory footprint ($\mathrm{relSize}_i$) and a custom-defined *importance* metric (I_i) to obtain the object's *moving factor* [61]:

$$\mathrm{MF}_i = \frac{S_i \cdot I_i}{\mathrm{relSize}_i} \qquad \text{where} \quad I_i = \frac{\mathrm{TotL}_i}{\sum_{o \in \mathrm{Objects}} \mathrm{TotL}_o} \tag{4.7}$$

TotL_i is the cumulative latency incurred by all memory accesses to i. Thus, the importance of object i is the total latency that accessing i caused, relative to the cumulative access latencies throughout the entire program.

ProfDP presents these moving factors to the programmer. While providing the hint that the object with the highest moving factor is the "top candidate" [61] to be placed into the faster memory, ProfDP ultimately lets the programmer decide the placement of each object. Thus, it is solely at their discretion to what extent they follow ProfDP's guidance. Such a user-curated approach greatly differs from H2M's and ecoHMEM's knapsack-based optimization.

4.3.2. User Experience

In contrast to H2M and ecoHMEM, ProfDP only automates **steps 1 and 2** of its workflow, the profiling and metric calculation step. For these steps, Wen et al. extended the HPCToolkit suite [1] to provide the necessary tools.

However, this extended version of HPCToolkit is not available anywhere. Jordà et al. [27] encountered the same issue when they wanted to compare their ecoHMEM framework to ProfDP. Thus, any programmer wanting to use ProfDP must fully reimplement its methodology, which is the route Jordà et al. and I took. Apart from the effort such a reimplementation requires, the original paper (which is the only available documentation on ProfDP) contains multiple ambiguities that impede a faithful reimplementation. For example, it remains unclear whether to use the latencies from the first or second run to compute each object's importance metric.[4]

Yet, even if ProfDP's original implementation were available, it would still have shortcomings in its user experience. For example, Wen et al. do not discuss how the user could apply ProfDP to heterogeneous memory systems with more than two memory types. Further, even if the system only has two memory types, the user may still be unable to choose between ProfDP's optimization approaches. This is because Wen et al.'s assumption that the slower memory type has *either* higher latency *or* lower bandwidth is false for, e.g., DRAM + NVM systems (cf. Section 2.1).

ProfDP also requires two profiling runs instead of one, causing significantly higher overheads than H2M or ecoHMEM. This issue is exacerbated by the second run in the bandwidth-aware approach (using only one core) likely being orders of magnitude slower than running the application normally. Further, running the program with only one core might alter its allocation and memory access behavior, yielding inaccurate results. Wen et al. also fail to address hardware capability issues. For example, in the latency-aware approach, each NUMA domain on its own might have insufficient space to host all the application's data. Also, the latency data that ProfDP requires might be unavailable on some platforms; latencies for writes in particular are not available in modern systems [27, 33], decreasing ProfDP's accuracy.

After (somehow) obtaining the moving factors, **steps 3 to 5** of ProfDP's workflow are fully manual. In **step 3**, the programmer must sort through the moving factors and decide which objects to place into the faster memory type. For real-world applications, this entails sorting through hundreds or thousands of allocation sites, where the programmer must consider the moving factor and size of each object.

Steps 4 and 5 require the programmer to facilitate the placements they decided upon. Wen et al. leave open whether the programmer should use an interception tool such as FlexMalloc or modify the source code with libraries such as `memkind` for this purpose. However, both cases require manual intervention since ProfDP does not generate FlexMalloc location files or similar artifacts. This non-negligible programmer input ProfDP requires at *every* step in its workflow is vastly detrimental to its user experience, with missing documentation worsening this issue.

[4] I tried reaching out to Wen et al. to answer this and other questions regarding ProfDP. However, my e-mail asking these questions remains unanswered to this day.

5. Evaluation and Comparison

As highlighted in Chapter 3, previous work has a shortage of thorough comparisons of different placement optimization frameworks. To address this shortage, I evaluate and compare the three frameworks presented in Chapter 4 under the aspects of *application performance*, *energy consumption*, and *user experience*. Further contrasting previous work, I employ a custom synthetic benchmark alongside proxy applications to compare the frameworks' optimization algorithms *in-depth*.

In this chapter, I first present the overall testing setup (Section 5.1) as well as the design of my synthetic benchmark along with the chosen proxy applications (Section 5.2). Using these applications, I conduct a set of experiments designed to each test different aspects of the frameworks' performance and energy optimization capabilities (Section 5.3). While running my tests, I experienced the workflow of each framework firsthand. Combined with user experience-related tests, this allows me to evaluate the practical hardships associated with using each framework (Section 5.4).

5.1. Testing Setup

I ran all tests on a DRAM + NVM heterogeneous memory system from RWTH Aachen University. Its specs are listed in Table 5.1. The memory modules are set up in an *interleaving* manner, i.e., one 32GiB DRAM module and one 128GiB NVM module are attached to each memory channel. Further, NVM is exposed as a separate NUMA domain instead of as a block device, and hyperthreading is disabled. I limited all experiments to use only one CPU socket with the closest DRAM and NVM NUMA domains via the `numactl` tool.[1] This serves to eliminate NUMA effects from the benchmarks to increase the reproducibility of the test results.

The system runs Rocky Linux version 8.10. All applications are compiled with GCC 11.3.0 and the flags `-g -O3 -march=native -fopenmp`. To further improve result consistency, I bind the OpenMP threads to physical cores by using `OMP_PLACES=cores` and `OMP_PROC_BIND=close` as environment variables. Timing measurements

[1] This excludes ProfDP's second profiling runs in the latency-aware approach. These are limited to using a remote NUMA memory domain as discussed in Section 4.3.1.

Parameter	Processor	Cores	Memory (DRAM)	Memory (NVM)
Specification	2× Intel Xeon Gold 6338 ("Ice Lake")	32 per socket (total: 64) @ 2.00GHz	16 × 32GiB = 512GiB DDR4 DRAM (8× per socket)	16 × 128GiB = 2TiB Intel Optane PMem (8× per socket)

Table 5.1.: Specifications of the used DRAM + NVM system

are taken within the applications themselves, while energy measurements use the LIKWID toolchain [59] version 5.3.0 (using `likwid-perfctr -g ENERGY`). I measure time and energy separately to avoid LIKWID's overheads skewing results.

All tests for the H2M framework use NumaMMA (commit `73ecec43`) for the profiling step. H2M's runtime (commit `da465fab`) and custom FlexMalloc fork (commit `6b4215a6`) then execute the optimized placements. The placement optimizer (commit `1116a390`) already includes configuration files for the used DRAM + NVM system since Klinkenberg et al. [33] ran their tests on the very same system. Hence, I use this configuration file instead of running step 1 of H2M's workflow again.

The tests for ecoHMEM use Extrae version 4.0.3 (commit `dad4f11e`) for the profiling step. Notably, version 4.0.3 is not the newest version of Extrae currently available but is the version explicitly linked to in the ecoHMEM repository. ecoHMEM's optimizer (commit `27e3be81`) already provides configuration files for a DRAM + Optane NVM system since Jordà et al. [27] used almost the same memory hardware as my test system in their tests. I do not modify these configuration files and their weighting coefficients. For executing ecoHMEM's data placements, I use H2M's customized FlexMalloc fork instead of a non-custom FlexMalloc + `memkind`. This is because `memkind` did not recognize the system's NVM, irrespective of whether it was exposed as a NUMA domain or block device. However, the memory allocators (cf. Section 2.3) H2M's FlexMalloc provides did recognize the system's NVM.

ProfDP's tests use my custom reimplementation of the ProfDP workflow.[2] Said reimplementation uses the Extrae profiler and H2M's custom FlexMalloc fork.

For all frameworks, I configured FlexMalloc to place objects into NVM if it cannot find placement decisions for the object. This can occur for objects whose lifetime is so short that they were not profiled by the sampling-based profilers.

5.2. Applications

As mentioned in this chapter's introduction, I evaluate the frameworks using not only proxy applications but also a synthetic benchmark. To elicit why I chose this approach and what benefits it brings over using proxy applications alone, I am going to present the design of my synthetic benchmark and the chosen proxy applications.

5.2.1. Synthetic Benchmark

My synthetic benchmark design serves two goals. First, it is highly configurable in the memory access behaviors discussed in Section 2.1 such that it can cover *all possible combinations* of these behaviors. Second, it is simple enough such that for each access behavior combination I can test which placement decisions are optimal via brute force. In combination, these two goals allow me to (a) test when each framework makes optimal placement decisions and (b) if they make non-optimal decisions, determine which access behavior(s) caused this non-optimal choice.

[2] Available under `https://git-ce.rwth-aachen.de/sesam120063/profdp-reimpl`

Algorithm 1 Synthetic benchmark

```
1:  int *p_1, *p_2, *p_3, *crit_obj
2:  Allocate & parallely initialize p_1, p_2, p_3, crit_obj with writes
3:  for iter ∈ {1, ..., NumRepeats} do
4:      pStart, pEnd ← PartitionArrays(p_1, p_2, p_3, Threads)
5:      cStart, cEnd ← PartitionArray(crit_obj, Threads + AddThreads)
6:      parallel for i ∈ [pStart, pEnd), j ∈ [cStart, cEnd); i += 8, j += 8 do
7:          GenerateAccesses(p_1, i); GenerateAccesses(p_2, i);
              ↪ GenerateAccesses(p_3, i)
8:          GenerateAccesses(crit_obj, j)

9:  function GenerateAccesses(array, baseIdx)
10:     idx1, ..., idx8 ← GetIndices(array, AccessPattern(array), baseIdx)
11:     AccessArray(array, WriteRatio(array), idx1, ..., idx8)
```

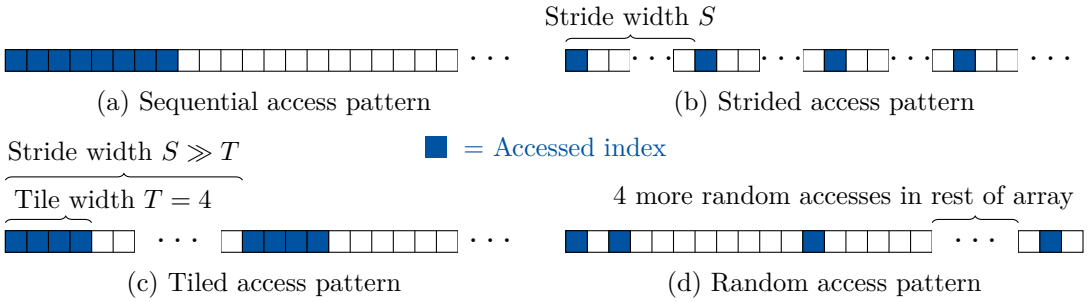


Figure 5.1.: Access patterns supported by the GenerateAccesses function

Algorithm 1 shows the **general structure** of my synthetic benchmark. It contains four objects: three *placeholder objects* (labelled `p_1`, `p_2`, and `p_3`) and one *critical object* (labelled `crit_obj`). The idea is to have the placeholder objects be "distractions" for the frameworks, which are accessed in the same, non-performance-critical manner across all configurations of the benchmark. The critical object, on the other hand, is the object for which I alter the access behaviors as previously described, making them less and less favorable for the slower NVM.

In the tests, I then give each framework only enough DRAM space to hold one of the four objects. By altering the accesses to the critical object, I can thus determine when each framework hits a decision boundary, i.e., when it deems the accesses to `crit_obj` so performance-critical as to warrant placing it into DRAM instead of NVM. Further, since the benchmark comprises only four objects, I can easily test which object should be placed into DRAM to yield optimal performance and/or energy consumption for each configuration. Hence, I can check whether the frameworks' decision boundary corresponds with the optimal decision boundary.

To discuss the **details** on how I implement this strategy, consider again the benchmark's pseudocode in Algorithm 1. The most important part of the benchmark is the GenerateAccesses function (lines 9–11). Drawing inspiration from the Hopscotch [2] and Mess [17] benchmarking suites, this function generates eight memory accesses at a time with a configurable access pattern and read-write ratio. Thus, GenerateAccesses covers two of the three access behavior *dimensions* discussed

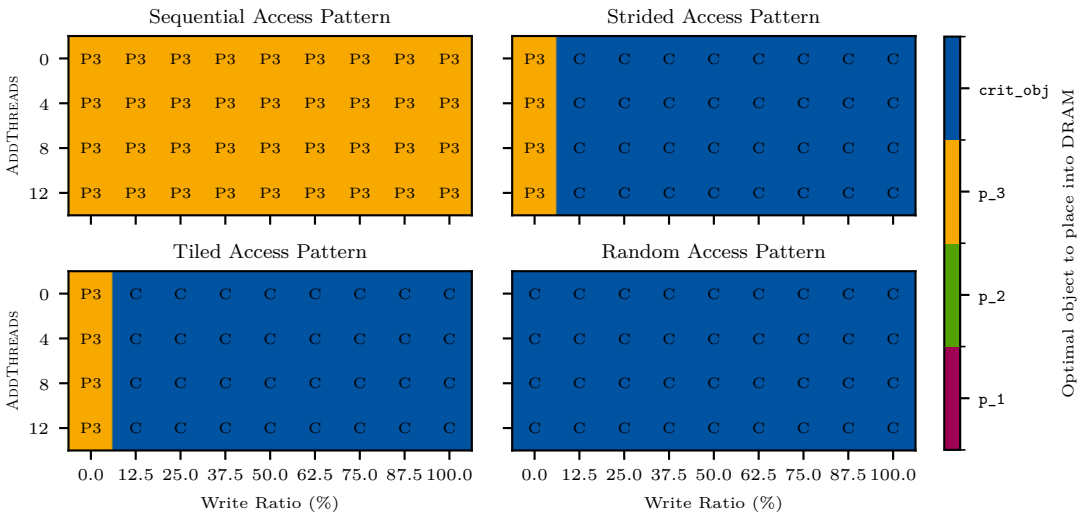


Figure 5.2.: Optimal object to place into DRAM to minimize execution time

in Section 2.1. It supports four different access patterns (shown in Figure 5.1) and can vary the WriteRatio in steps of 12.5 %.

To have the placeholder objects be the non-performance-critical "distractions" I intend them to be, the WriteRatio for `p_1`, `p_2`, and `p_3` is 0 %, i.e., they are never written to except for their initialization. This is because any writes to NVM vastly diminish its performance (cf. Section 2.1.2), meaning that writing to the placeholder objects would cause them to become performance-critical. Further, since irregular access patterns also considerably decrease NVM's performance, I configured the placeholder objects to have regular, cache-friendly access patterns. Specifically, `p_1` is accessed sequentially, `p_2` in a strided fashion, and `p_3` in a tiled manner. I chose different access patterns for the placeholder objects to not trivialize the frameworks' decision process. If all placeholder objects were accessed the same, this would reduce the decision process to a mere binary decision between putting the critical object or one of the *indistinguishable* placeholders into DRAM.

The third dimension discussed in Section 2.1, thread-level contention, can be configured via the AddThreads constant used in the array partitioning in lines 4–5. All objects are accessed in parallel by a baseline number of threads (in my tests I chose Threads = 8), with `crit_obj` having more parallel accessing threads depending on the value of AddThreads. Note that AddThreads does not change the overall number of accesses to `crit_obj` but distributes the same number of accesses across more threads, thereby yielding higher thread-level contention for it.

For ease of **notation**, I am going to denote each *configuration* of my benchmark, i.e., each combination of access pattern, read-write ratio, and thread-level contention used for the critical object's accesses, as a 3-tuple (a, x, y). a denotes the access pattern, $x \in [0, 100]$ is the write ratio in percent, and $y \in \{0, 4, 8, 12\}$ is the used AddThreads constant. In case the access pattern is obvious from the context, I omit it from the tuple (yielding the 2-tuple (x, y)).

To **verify** that my setup achieves the desired effects, I conducted baseline tests for

the synthetic benchmark. For each benchmark configuration, I placed each object into DRAM individually and measured the resulting execution time and energy usage. From these measurements, I can determine the optimal object to place into DRAM to minimize execution time or energy consumption.

Figure 5.2 shows the optimal placements for minimizing execution time. Each tile in each heatmap corresponds to one benchmark configuration, with the color and label indicating which of the four objects should be placed into DRAM to minimize execution time. For example, the leftmost-uppermost tile in the heatmap titled "Sequential Access Pattern" shows that to minimize the execution time of the $(\text{Sequential}, 0, 0)$ configuration, `p_3` should be placed into DRAM.

In most configurations, placing `crit_obj` into DRAM is the optimal decision, showing that my design succeeded in having the placeholder objects only be non-performance-critical distractions most of the time. Only two exceptions break this rule. First, if `crit_obj` is accessed sequentially, placing `p_3` into DRAM is always the optimal decision irrespective of the writes to the critical object. These results corroborate Yang et al., who report that NVM "can efficiently handle small stores, if they exhibit sufficient locality" [63]. Second, if `crit_obj` is not written to, the access pattern becomes the main performance-influential dimension. Therein, `p_3`'s tiled access pattern is more performance-critical than the sequential and strided access patterns of `crit_obj` in the top heatmaps. Further, even if `crit_obj` is accessed in a tiled fashion, placing `p_3` into DRAM instead is still the optimal choice. This indicates that increasing the thread-level contention on `crit_obj` does not outweigh the runtime decrease caused by distributing the accessing work over more threads.

These placements minimizing execution time also minimize the *energy consumption*, since the differences in power draw between the different placement options are minimal. Thus, the energy usage (which equals Power $\times$ Time) is dominated by the runtime. The interested reader may find the exact timing, energy, and power data to verify these claims in Appendix A. Said appendix also discusses how my baseline results corroborate the existing hardware studies cited in Section 2.1.

5.2.2. Proxy Applications

For my tests, I further chose the following four proxy applications:

- **CloverLeaf** [38]: structured grid Lagrangian-Eulerian hydrodynamics code
- **Ligra** [54]: shared memory graph processing framework
- **LULESH** [28]: hydrodynamics simulator on unstructured grids
- **XSBench** [58]: neutron simulation code using Monte Carlo methods

I chose these specific proxy applications for two reasons. First, they cover a wide range of computational patterns and application domains, making my results as generalizable as possible. To achieve this, I oriented myself on the Berkeley dwarf taxonomy [5] and picked applications from different dwarfs. This approach differs from previous work, which often chose applications too similar to one another for

Application	Parameters/Input	Memory Footprint	# Allocated Objects
CloverLeaf	`bm64_short` input file	11811 MB	38
Ligra (PageRank algorithm)	Graph generated with `./rMatGraph 12000000`	5634 MB	24
LULESH	`-s 192 -i 50`	6524 MB	6714
XSBench	`-t 32 -s large -l 1024`	5923 MB	364

Table 5.2.: Chosen proxy applications and inputs

their evaluation. For example, Jordà et al. [27] evaluate ecoHMEM separately on the LULESH, HPCG [22], and miniFE [36] proxy applications despite them having a very similar computational pattern [22, 47]. Second, these proxy applications cover a wide range in the number of objects they allocate, as Table 5.2 shows. Thus, I can test how the number of objects in an application affects the frameworks' efficacy.

In combination, the synthetic benchmark and the named proxy applications enable a "symbiotic" evaluation approach. The detailed results from the synthetic benchmark are going to assist in understanding the results for the proxy applications. On the other hand, the results for the proxy applications will cross-validate the relevance of the insights generated from the synthetic benchmark results.

5.3. Performance Experiments & Results

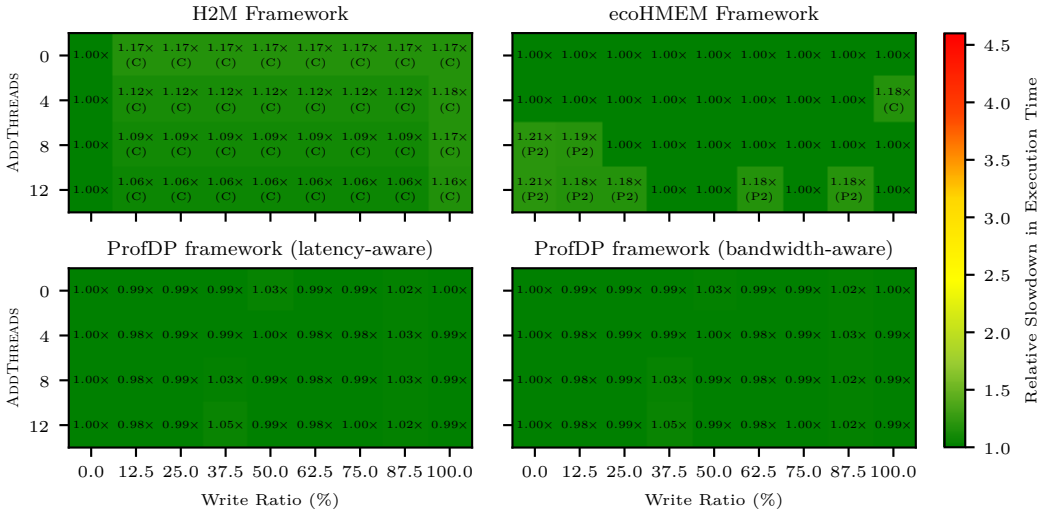
To evaluate the frameworks' performance and energy optimization capabilities, I conducted three experiments. The first two experiments run the frameworks on my synthetic benchmark, and the third experiment runs them on the four proxy applications. In the following, I present each experiment, its setup, and its results.

5.3.1. Synthetic Benchmark (DRAM Profiling)

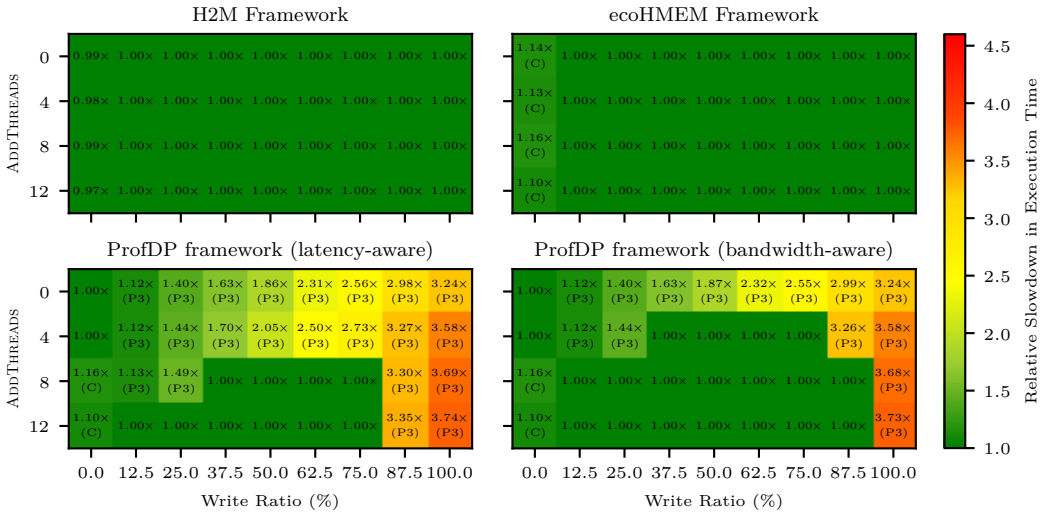
The first experiment's **setup** is the following: The frameworks profile each configuration of the synthetic benchmark once. During this profiling step, all application data is placed into *DRAM*. Since DRAM is relatively invariant to `crit_obj`'s changing access behavior dimensions compared to NVM (cf. Section 2.1), the gathered data is not skewed by any hardware sensitivities. Therefore, the taken placement decisions serve as a baseline to evaluate each framework's efficacy, meaning this first experiment can answer the following research questions:

1. In *how many configurations* do the frameworks make the optimal decision?
2. If the frameworks make non-optimal decisions: How *consequential* are they?
3. What *systemic issues* underly each framework's optimization algorithm?

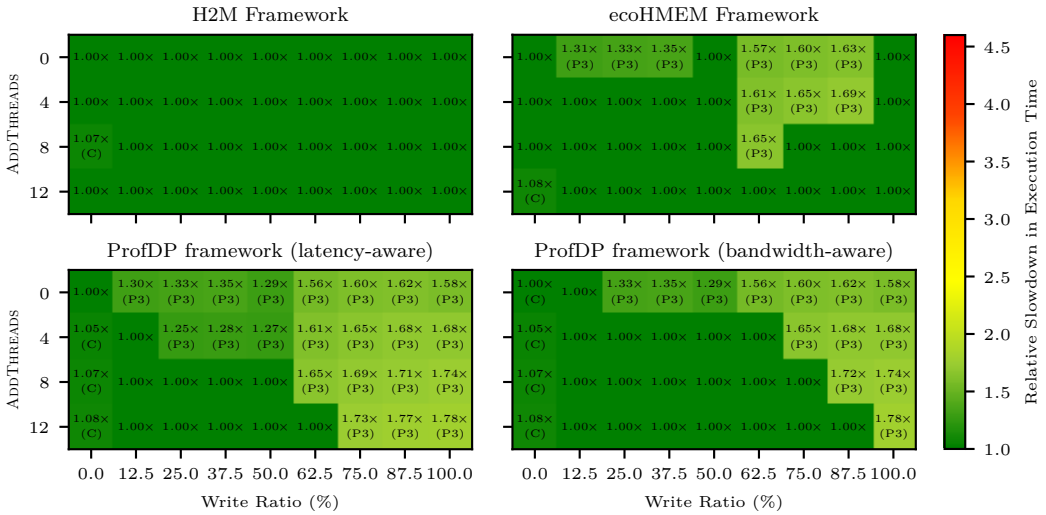
To answer the second question, I run each framework's placement decisions five times per benchmark configuration to measure their average execution time and energy usage. Combining these measurements with the baseline tests conducted in Section 5.2.1, I can compute the *slowdown/overhead* of any non-optimal decision.



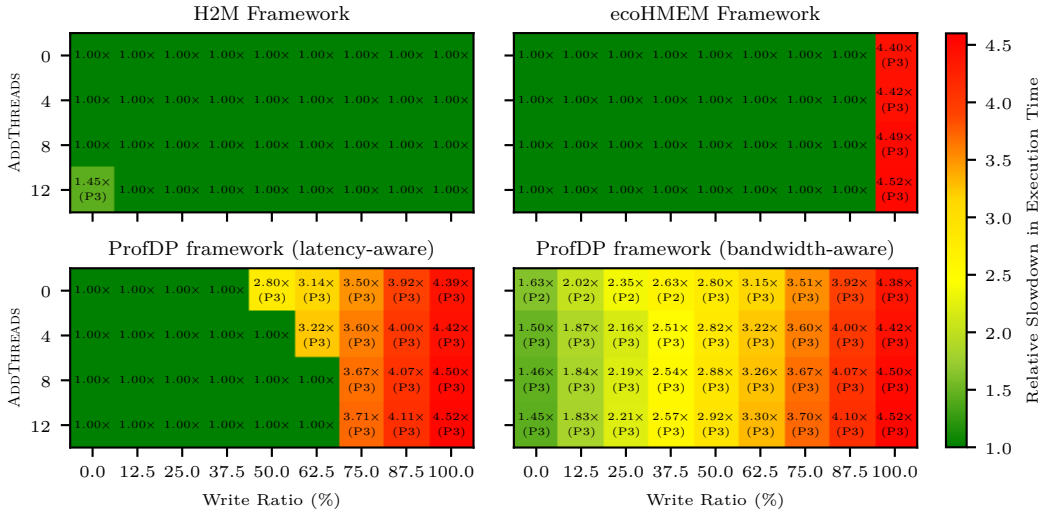
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure 5.3.: Slowdown in execution time of the frameworks' placement decisions over the optimal placement decisions

Figure 5.3 shows the **results** of this experiment. Each subfigure contains four heatmaps, one for each framework's results for the configurations accessing `crit_obj` with a certain access pattern. Each tile shows the slowdown of the frameworks' decisions over the optimal decisions as determined in the baseline tests. If any framework made a non-optimal decision in a configuration, the object that the framework placed into DRAM instead is shown in braces below the numeric slowdown. Note that ProfDP has two heatmaps per subfigure, one each for the latency- and bandwidth-aware optimization approach, since it is unclear which approach to use for DRAM + NVM systems (cf. Section 4.3.2). Further, H2M uses the IDP-LT knapsack model, since the synthetic benchmark has no distinct computational phases.

Since the optimal placement decisions for minimizing energy consumption are the same as for minimizing execution time (cf. Section 5.2.1), the energy consumption overheads of non-optimal decisions are similar to the shown slowdowns. Hence, I omit them here; the interested reader may find them in Appendix B.1.

For the benchmark configurations accessing `crit_obj` **sequentially**, Figure 5.3 (a) shows that both of ProfDP's optimization approaches always make the optimal placement decision, placing `p_3` into DRAM. ecoHMEM makes the same optimal decision most of the time, with H2M mostly making the non-optimal decision of placing the critical object into DRAM.

H2M's non-optimal decisions are *systemic* in that they only occur once `crit_obj` is written to. This is because H2M does not respect the access pattern of the writes. It does not change the bandwidth and latency metrics it uses in its optimization based on the access pattern, despite the access pattern significantly impacting NVM's write bandwidth and latency. Thus, H2M estimates that the writes to `crit_obj` are more performance-critical than the reads to `p_3`. However, this is false when taking the access pattern of both objects into account, as Section 5.2.1 showed.

In contrast, ecoHMEM does not make the systemic error of overvaluing `crit_obj`'s sequential writes. This is because ecoHMEM only uses cache misses in its placement optimization, with sequential writes causing virtually no such misses. However, focusing on cache misses alone leads to ecoHMEM placing `p_2` into DRAM instead of `p_3` in some configurations. The latter object has the higher cache miss rate of the two, which should cause ecoHMEM to place it into DRAM. Yet, ecoHMEM's *sampling-based* profiling sometimes records more cache misses for `p_2` than for `p_3` due to a low sampling rate (100 Hz) and a short profiling runtime (~15 seconds).

ProfDP always makes the optimal decision of placing `p_3` into DRAM in all configurations, assigning it a moving factor between 1.7× and 2× higher than any other object. This shows that ProfDP's approach of using latency data for its optimization can correctly identify the performance-critical nature of different access patterns.

Interestingly, the recorded slowdowns for ProfDP vary between 0.98× and 1.05× despite it making the optimal decision in all cases. Not all of this variation is attributable to run-to-run variance, since the maximum relative standard deviation is only 2.5 %. The slowdown variations having the same pattern in both of ProfDP's optimization approaches hint at a systemic issue; yet, even after extensively reviewing my testing setup, I cannot determine the cause of said issue.

For the benchmark configurations accessing `crit_obj` in a **strided** fashion, Figure 5.3 (b) shows that H2M now always makes optimal placement decisions. ecoHMEM also always makes optimal decisions except when `crit_obj` is only read, with ProfDP now being the worst of the three frameworks.

H2M's optimal decision-making is to be expected in light of the results for the sequential access pattern configurations. It, this time correctly, identifies the writes to `crit_obj` as performance-critical, leading to H2M assigning `crit_obj` a value 5× to 25× higher than any other object. When `crit_obj` is not written to, H2M correctly identifies `p_3` as the most performance-critical object due to its high cache miss rate. In contrast to ecoHMEM, H2M's profiler can recognize this higher miss rate because of its higher sampling frequency (6000 Hz).

ecoHMEM places `crit_obj` into DRAM in all configurations due to its strided access pattern causing significant amounts of cache misses. In contrast to the sequential access pattern configurations, ecoHMEM never places `p_2` or `p_3` into DRAM despite their similar cache miss rates. This is because some of `crit_obj`'s cache misses are write misses (at least in the configurations with a write ratio > 0 %), which ecoHMEM values higher due to the chosen weighting coefficients (cf. Section 4.2.2). However, even when the critical object is only read, ecoHMEM still places it into DRAM instead of `p_3`. In the $(0, 0)$- and $(0, 4)$-configurations, this is again caused by ecoHMEM's low sampling frequency not recognizing `p_3`'s slightly higher cache miss rate. Yet, when AddThreads increases above 4, cache misses for the critical object increase significantly due to higher cache contention, which ecoHMEM correctly profiles. Thus, ecoHMEM's non-optimal decisions for these configurations demonstrate the *systemic* issue of focusing on cache misses alone. Note that this is *not* an issue of ecoHMEM's bandwidth-aware object classification, since that never altered any of ecoHMEM's decisions in this entire experiment.

ProfDP makes noticeably more non-optimal decisions than H2M and ecoHMEM. Focusing on the configurations where `crit_obj` is only read, ProfDP at first makes the optimal decision of placing `p_3` into DRAM for the $(0, 0)$- and $(0, 4)$-configurations. Yet, it then shifts to placing `crit_obj` into DRAM for higher AddThreads values. Similar to ecoHMEM's results, this is due to `crit_obj`'s cache misses increasing alongside thread-level contention. More cache misses cause higher cumulative latencies, thus increasing `crit_obj`'s *importance metric* and, by extension, its moving factor above that of `p_3`. The same pattern of more cache misses increasing the critical object's importance continues when it is written to. However, when `crit_obj` is written to, placing it into DRAM now becomes the optimal decision, explaining ProfDP's better performance for higher AddThreads values. Yet, once the write ratio increases over 75 %, `crit_obj`'s importance metric deteriorates due to missing latency information for writes (cf. Sections 2.2 and 4.3.2). Accordingly, ProfDP places `p_3` into DRAM instead of `crit_obj` in these configurations.

Notably, ProfDP's bandwidth-aware optimization approach yields optimal placement decisions in more cases than its latency-aware approach. That is because increasing AddThreads raises not only `crit_obj`'s importance metric but also its *bandwidth sensitivity* metric. In combination, this leads to ProfDP placing `crit_obj` into DRAM for lower AddThreads values, as Figure 5.3 (b) shows.

For the **tiled** access pattern configurations, Figure 5.3 (c) shows that H2M performs the best out of the frameworks, making only one non-optimal decision. ecoHMEM makes more non-optimal choices, yet still considerably fewer than ProfDP.

H2M's one "slip-up" for the $(0, 8)$-configuration is down to variance in the sampling-based profiling. In this specific configuration, H2M profiled ~1000 more writes from `crit_obj`'s initialization than in the other configurations. This caused its object value to be larger than `p_3`'s, leading to the change in placement decisions.

ecoHMEM's results are a combination of its previously discussed focus on cache misses and its low-frequency sampling. For low AddThreads values, its profiler measures significantly more cache misses for `p_3` than for `crit_obj`, leading to the corresponding placement decisions. For higher AddThreads values, the critical object's cache misses become more than `p_3`'s, changing ecoHMEM's placement decisions. Noteworthy is that ecoHMEM places `p_3` into DRAM at all in configurations where the critical object is written to. The cause for this is that ecoHMEM profiles 10× fewer write cache misses than read misses for `crit_obj`. In combination with the low relative difference between the read and write miss weighting coefficients, this causes `p_3`'s higher read cache miss numbers to outweigh the critical object's write misses. Outliers in this regard are the configurations with write ratios of 50 % and 100 %, where ecoHMEM profiles drastically more write cache misses (> 2000 write misses compared to 500–800).

ProfDP's results follow the same pattern as its results for the strided access pattern configurations, for exactly the same reason (i.e., `crit_obj`'s varying importance metric). However, contrary to the results for the strided access pattern configurations, `p_3`'s latency sensitivity metric is also higher than that of the critical object. Further, `crit_obj`'s latency sensitivity now decreases when increasing

AddThreads, explaining why ProfDP's latency-aware approach places `p_3` into DRAM even for higher AddThreads values. The root cause for this is the *relative nature* of the aforementioned sensitivity metrics. Increasing thread-level contention causes `crit_obj`'s absolute average access latency to rise above `p_3`'s (600 vs. 400 CPU cycles for the local NUMA domain run). Yet, switching from the local to the remote NUMA domain in the second run causes a 200 cycle latency increase for both objects, meaning the relative difference between runs is lower for `crit_obj` than for `p_3`. The bandwidth-aware approach does not suffer from the same problems, hence performing better than the latency-aware approach.

For the **random** access configurations, Figure 5.3 (d) shows that H2M again has one slip-up when `crit_obj` is only read. In contrast, ecoHMEM makes its non-optimal decisions when the critical object is only written to. ProfDP again makes the most non-optimal decisions of all frameworks in both optimization approaches.

H2M's non-optimal decision for the $(0, 12)$-configuration is notable in that the critical object's random access pattern has a higher cache miss rate than `p_3`'s tiled access pattern, which should cause H2M to place `crit_obj` into DRAM. However, for this configuration, H2M profiled ~2000 fewer read cache misses for `crit_obj` than for `p_3`. Since the other configurations only reading the critical object do not show the same behavior, this is again a profiling outlier rather than a systemic issue.

ecoHMEM's optimal decisions for write ratios ≤ 87.5 % are due to `crit_obj`'s random access pattern causing the most cache misses. Yet, the number of profiled write cache misses for the critical object is again significantly lower than the number of profiled read misses (2000 write misses vs. 8500 read misses). Once the write ratio is set to 100 %, the read misses disappear, leading to `crit_obj` only having 2000 cache misses in comparison to `p_3`'s 6100. Since the relative difference between the read and write miss weighting coefficients is only $\frac{10}{5} = 2$, `p_3`'s object value is thus larger than `crit_obj`'s. This shows the *systemic* issue of using arbitrary weighting coefficients, especially considering the large slowdown of the non-optimal decisions.

ProfDP's results for the latency-aware approach follow the exact same pattern as the results for the tiled access pattern configurations for exactly the same reasons discussed there. Yet, its bandwidth-aware approach now performs worse than its latency-aware approach, in contrast to said previous results. The reason behind ProfDP making only non-optimal decisions in this approach is that `crit_obj`'s bandwidth sensitivity metric, and thus, moving factor, are very low. Specifically, `crit_obj`'s moving factor is only ~30, while `p_2` and `p_3` have moving factors in the interval $[90, 200]$. However, this result makes sense, as the random access pattern is the least bandwidth-bound of the four access patterns.

5.3.2. Synthetic Benchmark (NVM Profiling)

The **setup** of the second experiment is the following: The frameworks again profile each synthetic benchmark configuration once. Yet, this time, the application's data is placed into *NVM* during said profiling. This setup mirrors the most likely usage scenarios for heterogeneous memory placement optimizers, where the faster memory

is not going to have enough space to host all application data for the profiling step. Since NVM is significantly affected by `crit_obj`'s changing access behaviors, this experiment is going to answer the research question of how these hardware characteristics are going to *change* the frameworks' decision-making.

Figure 5.4 shows the **results** for this experiment. Except for the change of using NVM for the profiling step instead of DRAM, the test setup is exactly the same as in Section 5.3.1. Continuing these similarities, I show only the execution time slowdowns since the energy consumption overheads are similar to the shown slowdowns. The interested reader may find these plots in Appendix B.2.

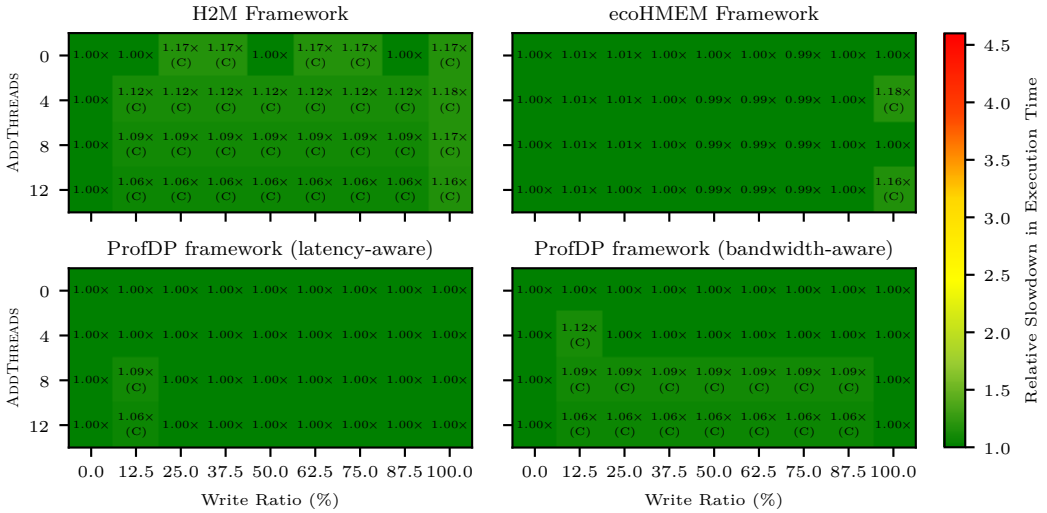
For the **sequential** access pattern configurations, Figure 5.4 (a) shows significant changes in the placement decisions for ecoHMEM and ProfDP. H2M, on the other hand, makes almost the same decisions as in Section 5.3.1.

ecoHMEM improves over the DRAM profiling experiment, now only making two non-optimal decisions instead of eight. The reason behind this change is that NVM's lower overall performance causes the profiling run to take significantly longer (~60 seconds instead of 15). Thus, even ecoHMEM's low-frequency sampler can now recognize `p_3`'s higher cache miss rate and place it into DRAM more often. The two non-optimal decisions for the $(100, 4)$- and $(100, 12)$-configurations are the consequence of profiling variance. In these configurations, ecoHMEM profiled significantly more write cache misses for `crit_obj` than in any other configuration.

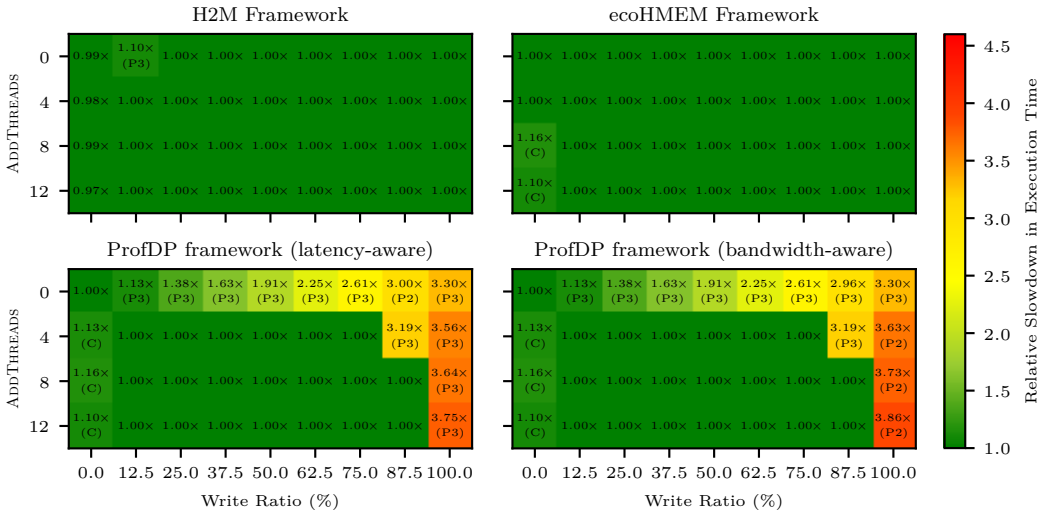
ProfDP, on the other hand, now makes non-optimal decisions for this set of configurations, especially in the bandwidth-aware approach. This is because NVM's high sensitivity to thread-level contention causes `crit_obj`'s bandwidth sensitivity metric to be significantly (sometimes over 10×) higher than in the DRAM profiling. Writes in particular exacerbate this sensitivity to thread-level contention, explaining why the non-optimal decisions only start when the critical object is written to. The latency-aware approach makes two mistakes (instead of none in DRAM profiling) for much the same reason. NVM's high contention sensitivity also raises `crit_obj`'s latency sensitivity metric with higher AddThreads values, bringing its moving factor much closer to (but still below) that of `p_3`. In the two specific configurations where ProfDP now makes non-optimal decisions, the critical object's access latencies spiked by pure coincidence, causing its moving factor to exceed that of `p_3`.

H2M's placement decisions see almost no changes over the DRAM profiling, containing the same systemic error of overvaluing the critical object's sequential writes. The changes in placement decisions in the $(25, 0)$-, $(50, 0)$-, and $(87.5, 0)$-configurations are a product of the sampling-based profiling. In these configurations, NumaMMA happened to profile significantly more write cache misses from `p_3`'s initialization than for any other object. Thus, H2M reports 37 % more write cache misses for `p_3` than for `crit_obj`, despite the latter receiving writes in the "main computation" of the benchmark. Similar anomalies do not occur for configurations with AddThreads > 0, since the higher thread-level contention yields significantly more cache misses for the critical object, offsetting any profiling variance.

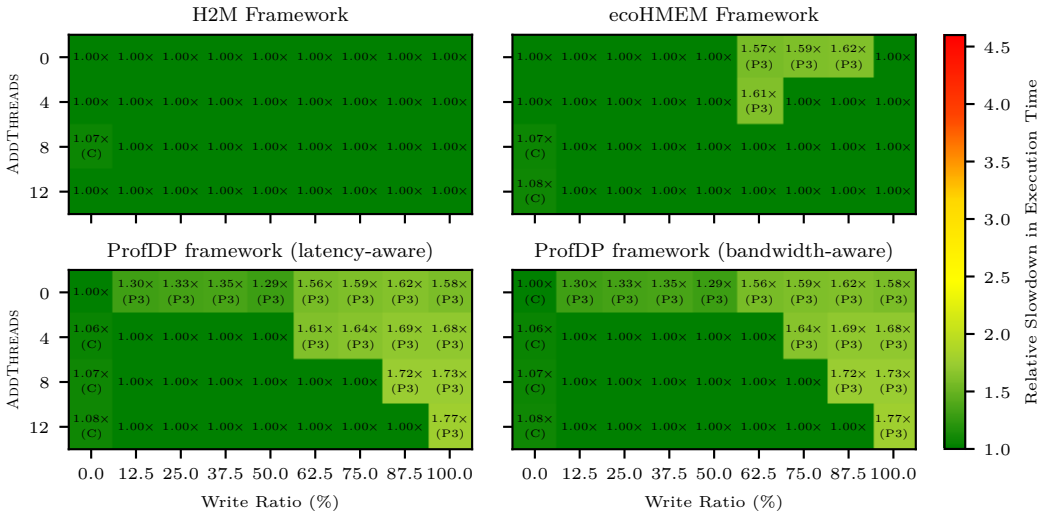
For the benchmark configurations accessing the critical object in a **strided** fashion, Figure 5.4 (b) shows significant changes over the DRAM profiling experiment



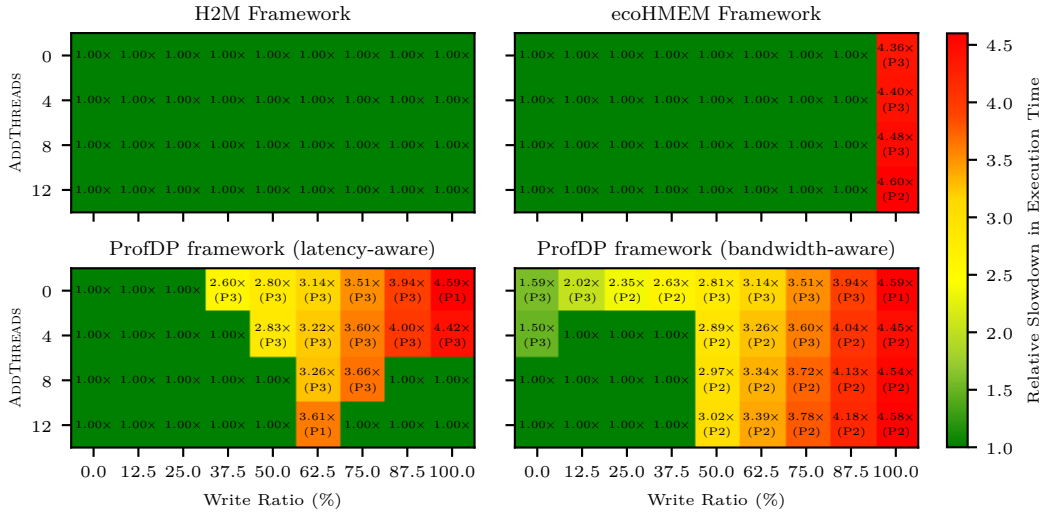
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure 5.4.: Slowdown in execution time of the frameworks' placement decisions over the optimal placement decisions

only for the ProfDP framework. H2M and ecoHMEM alter their placement decisions in only one or two configurations, respectively.

ProfDP significantly benefits from the NVM profiling in this set of configurations, with both approaches now performing equally well. The reason for this is ProfDP's importance metric. As explained in Section 5.3.1, `crit_obj`'s importance metric increases along with the AddThreads constant. When profiling on NVM, this increase is more pronounced due to NVM being more sensitive to thread-level contention than DRAM. Thus, ProfDP places the critical object into DRAM even for configurations with low AddThreads values in both optimization approaches.

H2M, in contrast, now makes a single non-optimal decision for the $(12.5, 0)$-configuration, again due to its sampling-based profiling. NumaMMA, seemingly systematically, now profiles more write cache misses from `p_3`'s initialization in all configurations. This brings `p_3`'s object value closer to `crit_obj`, making it more likely for a profiling outlier to cause a change in placement decisions.

ecoHMEM slightly improves over the DRAM profiling, now making optimal decisions for the $(0, 0)$- and $(0, 4)$-configurations. This behavior has the same underlying reason as the sequential access pattern results: The longer profiling time allows ecoHMEM to more accurately recognize `p_3`'s slightly higher cache miss rate in comparison to `crit_obj`. Yet, once AddThreads increases, the critical object's cache misses also increase, causing non-optimal decisions as in the DRAM profiling.

For the **tiled** access pattern configurations, Figure 5.4 (c) shows no changes in H2M's placement decisions compared to DRAM profiling, while ecoHMEM and ProfDP both improve significantly. ecoHMEM's and ProfDP's improvements occur for the reasons already discussed in the previous configuration sets. ProfDP benefits from a sharper increase in `crit_obj`'s importance metric, while the longer profiling time allows ecoHMEM to profile more write cache misses for the critical object.

Finally, in the **random** access pattern configurations, Figure 5.4 (d) shows only minute differences for H2M and ecoHMEM in comparison to DRAM profiling, while ProfDP sees significant changes to its placement decisions.

H2M fixes its non-optimal placement decision for the $(0, 12)$-configuration from the DRAM profiling experiment. This is because NumaMMA now profiles more writes from `crit_obj`'s initialization, causing H2M to assign it a higher value than `p_3`. Yet, as in the DRAM profiling, NumaMMA still records significantly fewer read cache misses for the critical object than for `p_3` in only this singular configuration.

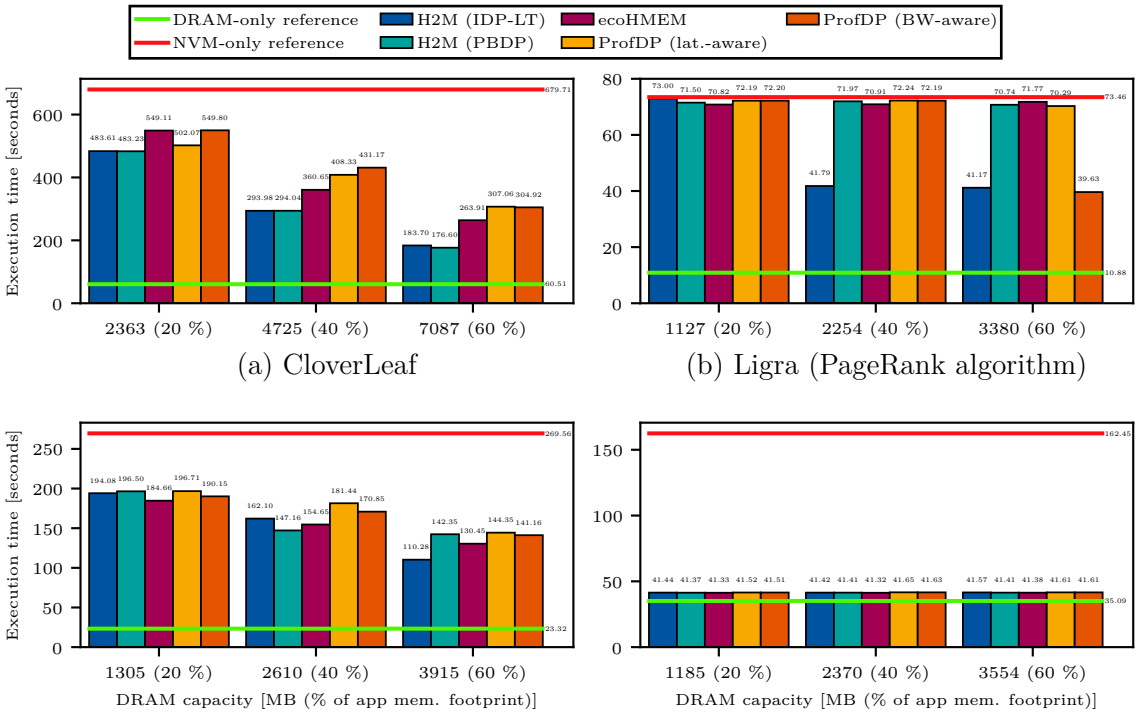
ecoHMEM still makes the same systemic error for the configurations with a write ratio of 100 % for exactly the same reasons discussed in Section 5.3.1. Interestingly, ecoHMEM changes its placement decision in the $(100, 12)$-configuration, now placing `p_2` into DRAM instead of `p_3`. This is because ecoHMEM profiles more cache misses for `p_2` than for `p_3` in this configuration. However, this does not change the fact that both placement decisions are non-optimal, causing vast slowdowns (~350 %) over the optimal placement decision.

ProfDP's results for the latency-aware approach are surprising. It now places `crit_obj` into DRAM even in configurations where it is only written to, even though ProfDP has no latency information on it in these configurations. This behavior is caused by an idiosyncrasy of my reimplementation of ProfDP. If an object has no available latency data, I assign it a default moving factor of 0. In the $(100, 8)$- and $(100, 12)$-configurations, the moving factors of the placeholder objects are *negative*, leading to `crit_obj` having the highest moving factor. The placeholder objects having negative moving factors is noteworthy, since this can only occur if their latency sensitivity metrics are negative. This, in turn, implies that their average access latencies were lower (!) when all data was placed into a remote NUMA domain instead of a local one. I reason that this counterintuitive behavior is caused by contention effects. Specifically, placing all application data into a remote NUMA domain still measurably increases the access latencies for `crit_obj`. This lowers the contention on NVM, allowing the placeholder object accesses to complete faster.

ProfDP's bandwidth-aware approach benefits from the NVM profiling, now making optimal placement decisions for 11 configurations instead of 0. This is because NVM's higher sensitivity to thread-level contention causes not only `crit_obj`'s importance metric but also its bandwidth sensitivity metric to increase alongside the AddThreads constant. However, once the critical object's write ratio exceeds 50 %, the bandwidth sensitivities of the placeholder objects also increase drastically, leading to ProfDP placing them into DRAM instead. I opine this is because the random writes to `crit_obj` stall the reads of the placeholder objects since NVM's write performance is much lower than its read performance. Thus, the access latencies to the placeholder objects increase as `crit_obj` is written to more and more.

5.3.3. Proxy Applications

For testing the proxy applications, I have the following **setup**: Each framework profiles each application once with all of its data placed in DRAM. Since the memory



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

Figure 5.5.: Execution time results for the proxy applications

footprints of all applications are smaller than the physically available DRAM space, I restrict the frameworks to use only 20 %, 40 %, and 60 % of the program's memory footprint as DRAM space to not trivialize the decision process. This setup will answer the research question of how the systemic weaknesses identified in the previous experiments *affect* the frameworks in *real-world contexts*. To that end, I run each framework's placement decisions for each application-capacity limit combination 10 times and measure their execution time and energy consumption.

Figure 5.5 shows the execution time **results** for this experiment. For reference, each plot also includes execution times when placing all application data into either DRAM or NVM. For H2M's phase-based optimization (PBDP), I split each application into phases based on the computational hotspots identified by Intel's VTune tracing tool. Therein, I tested setups with fewer and more phases, of which the result plots always show the best-performing setup.

Noteworthy is that the difference in power draw between the frameworks' placement decisions is minimal, similar to the synthetic benchmark. Hence, the runtime again dominates the energy consumption, meaning the energy consumption plots are almost identical to the shown execution time plots. As such, I do not show them here; they can be found in Appendix C.

For **CloverLeaf**, Figure 5.5 (a) shows that H2M performs best across all capacity limits. For the 40 % and 60 % limits, ecoHMEM is the second-best framework, while for the 20 % limit, ProfDP's latency-aware optimization outperforms ecoHMEM.

The reason behind ecoHMEM's and ProfDP's worse performance is that they

Object	Cache hits (R/W)	Cache misses[a] (R/W)	Cache hit rate (R/W)	Avg. access latency (CPU cycles)
`xarea`	50694/147	3297/201	93.89 %/42.24 %	35.76
`yarea`	56897/188	2356/187	96.02 %/50.13 %	25.7
`volume`	44027/230	6207/197	87.64 %/53.86 %	61

[a] Reported read misses are LLC misses, reported write misses are L1D misses

Table 5.3.: Select objects from CloverLeaf with their memory access metrics

place the `xarea`, `yarea`, and `volume` objects into DRAM, a decision which H2M never makes. Table 5.3 shows the access data for these objects as profiled by NumaMMA. Inspecting these metrics reveals that all three objects are accessed in manners performance-friendly for NVM. They have high read cache hit rates, with reads further making up a supermajority of the accesses. Thus, `xarea`, `yarea`, and `volume` are not as performance-critical as other objects within CloverLeaf, which have a lower cache hit rate and receive more writes. H2M correctly identifies this non-performance-critical nature, enabling it to place other, more performance-critical objects into DRAM, whereas ecoHMEM and ProfDP do not.

ecoHMEM's and ProfDP's non-optimal decisions for these objects are a consequence of the systemic weaknesses I identified in their optimization algorithms in Section 5.3.1. ecoHMEM only considers cache misses in its optimization, without any surrounding context such as the cache hit rate. While `xarea`, `yarea`, and `volume` have high cache hit rates, they also have high absolute cache miss numbers in comparison to other objects within CloverLeaf. Thus, ecoHMEM incorrectly prioritizes them in its placement optimization. ProfDP's suboptimal decisions are due to the relative nature of its sensitivity metrics. Table 5.3 shows that the average access latencies for all three objects are very low, further reinforcing the notion that they are not performance-critical. Yet, due to the relative nature of the sensitivity metrics, the small absolute latency increase between the profiling runs causes the sensitivity metrics for these objects to become very large.

H2M's phase-based optimization changed almost no placement decisions in comparison to IDP-LT, no matter the chosen phase granularity. That PBDP performs better than IDP-LT in the 60 % is only due to testing variance. In this capacity limit, PBDP's variance was abnormally large with a relative standard deviation of 1.9 %; in all other tests, the maximum relative standard deviation was 0.5 %.

For **Ligra's PageRank algorithm**, Figure 5.5 (b) shows that only H2M's IDP-LT and ProfDP's bandwidth-aware approach make substantial improvements over placing all of Ligra's data into NVM. Interestingly, H2M and ProfDP reach their similar performance levels through different placement decisions.

H2M gains performance over the all-NVM baseline by placing `FL` into DRAM, which is an object used during graph initialization. ecoHMEM places this object into NVM due to its low number of cache misses (70 misses in 6000 accesses), while ProfDP places it into NVM since its low access latencies yield a low importance metric. Yet, both frameworks miss that 50 % of `FL`'s accesses are writes. In this case, `FL`'s writes are performance-critical despite their cache-friendly nature because they contend with writes to other objects in the graph initialization. Hence, placing

`FL` into DRAM reduces write contention on NVM, improving NVM's performance. H2M, which heavily prioritizes written-to objects in its placement optimization (cf. Section 5.3.1), is thus the only framework to identify `FL`'s performance-critical nature. Its phase-based optimization, however, did not place `FL` into DRAM due to profiling variance, instead placing some other, unimportant objects into DRAM.

ProfDP's bandwidth-aware approach gains its performance not from placing one particular object, but the *group* of the `Sums`, `edges`, and `vertices` objects into DRAM. It thereby puts all objects on the critical path of PageRank's computation into DRAM, speeding up the algorithm substantially; placing *any one* of these objects back into NVM deteriorates performance to all-NVM levels. This demonstrates that data placement optimization can benefit from considering that real-world applications often access objects together in logical groups. In such cases, placing any *singular* objects from these groups into faster memory is not beneficial, as then the other object's accesses become the bottleneck on the computation path.

ecoHMEM's poor performance is again a product of its pure focus on cache misses. While it makes similar placement decisions to ProfDP's bandwidth-aware approach in that it places `edges` and `vertices` into DRAM, it critically does not place `Sums` or `FL` into DRAM because their accesses cause very few cache misses. ecoHMEM's bandwidth-aware classification step did not fix this issue, as it classified neither `Sums` nor `FL` into the Thrashing category (cf. Section 4.2.1) in spite of them having the highest bandwidth usage of any objects according to ecoHMEM.

For **LULESH**, Figure 5.5 (c) shows that ecoHMEM performs best for the 20 % capacity limit. In the other capacity limits, it is surpassed by either H2M's IDP-LT or PBDP, with ProfDP having the worst performance across the entire test.

The main challenge in LULESH is that it allocates and deallocates most of its objects every computational iteration; I will call these objects *temporary objects* from here on. Thus, LULESH makes over 3000 allocations but only has ~50 live objects at any single time. In theory, this should benefit H2M, as it is the only framework of the three properly equipped to handle allocations made in a loop (cf. Section 4.2.1). And, in fact, it does benefit H2M for the 60 % capacity limit test.

In this test, H2M places the `x8n` and `y8n` temporary objects into DRAM, which ecoHMEM and ProfDP never place into DRAM. The reason for this is that each instance of the objects (alive for one singular iteration) is accessed relatively often compared to other temporary objects, but relatively little in comparison to the *permanently* allocated/alive objects. Since Extrae, which both ecoHMEM and ProfDP use, does not accumulate accesses over all instances of the temporary objects (cf. Section 4.2.1), both frameworks are unable to identify `x8n` and `y8n` as performance-critical. Thus, H2M's ability to distinguish different instances of the same allocation callstack reveals itself as valuable for real-world, iterative applications.

H2M's phase-based optimization only performs on par with IDP-LT when using very fine-granular phases, with more coarse-grained phases performing significantly worse than all other frameworks. Further, PBDP only beats IDP-LT for the 40 % capacity limit. In this test, H2M's PBDP places the `fy_elem` and `fz_elem` temporary objects into DRAM by migrating other, less important permanent objects

into NVM after they were initialized. Placing these two objects into DRAM yields a 15–20 second improvement in execution time, fully explaining the gap between IDP-LT and PBDP. Yet, PBDP does not place `x8n` and `y8n` into DRAM for the 60 % capacity limit, leading to it performing significantly worse than IDP-LT.

ecoHMEM outperforms H2M's IDP-LT for the 20 % and 40 % capacity limits despite Extrae being ill-equipped to deal with loop allocations. This is because ecoHMEM makes different, more optimal placement decisions for the permanently allocated objects. This, in turn, is due to H2M overvaluing sequential writes as discussed in Section 5.3.1. LULESH's objects are all written to in very cache-friendly (presumably sequential) patterns, meaning these writes are non-performance-critical for NVM. Yet, H2M's optimization does not consider this access pattern, causing it to place these objects into DRAM. ecoHMEM's bandwidth-aware classification step helps solidify this advantage by classifying many temporary objects into the Streaming-D category, evicting them into NVM. Thus, despite Extrae's aforementioned weakness, ecoHMEM and H2M made almost identical placement decisions regarding the temporary objects. However, LULESH was the only tested application where ecoHMEM's bandwidth-aware step had any effect.

ProfDP's poor performance across all capacity limits is because it places two large objects, `p_old` and `q_old`, into DRAM. These objects are not performance-critical since they are accessed very little (only ~5000 accesses, compared to $\sim 15000 - 50000$ for other objects) and mostly in a sequential, reading fashion. Yet, in the profiling runs, access latencies for these objects spiked a few times by pure coincidence, producing latencies of ~600–1500 cycles instead of the usual 50–100 cycles. As these objects are accessed relatively little, such latency spikes skew the latency averages and thus the moving factors, leading to non-optimal placement decisions. This demonstrates that ProfDP relying on latency data alone for its placement optimization can be a weakness, especially for objects that are accessed relatively little.

ProfDP's bandwidth-aware approach does not place these objects into DRAM in the 20 % capacity limit test, since no latency spikes occurred in the bandwidth-aware profiling runs. Thus, ProfDP's performance in this test is much closer to ecoHMEM. Yet, for the 40 % and 60 % capacity limits, ProfDP again places `p_old` and `q_old` into DRAM, detrimenting its performance. The cause for this is the relative nature of the sensitivity metrics, identical to the results for CloverLeaf.

Lastly, for **XSBench**, all frameworks perform virtually identically across all capacity limits and almost reach the all-DRAM reference measurement, as Figure 5.5 (d) shows. This is because XSBench allocates only four objects with a size larger than 1 MB, only three of which are frequently accessed. Further, one array (`index_grid`) takes up over 80 % of the application's memory footprint, thus immediately making it too large for any of the 20 %, 40 %, and 60 % capacity limits. Therefore, the placement decisions in XSBench are trivial, with all frameworks placing the two remaining objects (`nuclide_grid` and `unionized_energy_array`) into DRAM. Coincidentally, these objects are the most performance-critical objects in XSBench, meaning their placement into DRAM yields almost all-DRAM performance.

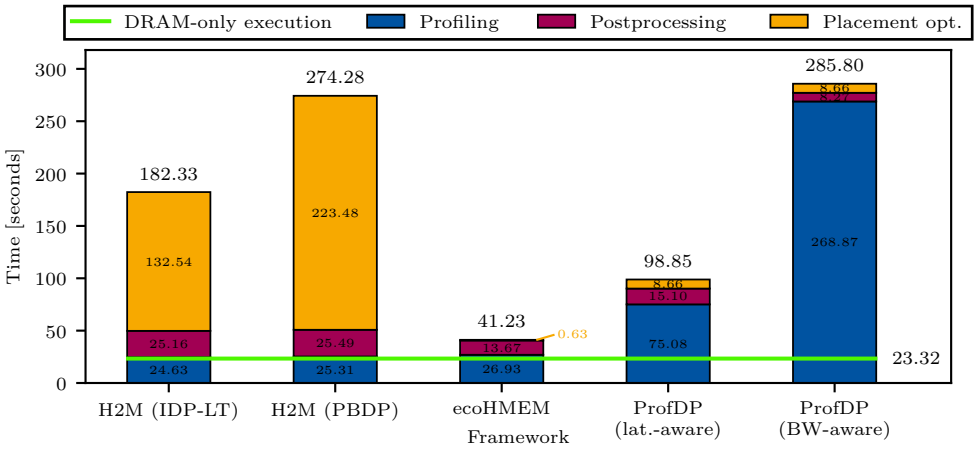


Figure 5.6.: Time taken by each framework to optimize the data placement for LULESH

5.4. User Experience Experiments & Results

My evaluation of the frameworks' user experience comprises two parts. First, in Section 5.4.1, I quantify the user-side overhead of applying each framework to a given application in terms of the time and effort the programmer needs to expend. Second, in Section 5.4.2, I discuss the various bugs and hardships I encountered when using the frameworks. Therein, I especially focus on how the user experience issues I discussed on a theoretical level in Chapter 4 affected my tests in practice.

5.4.1. Overhead of Framework Usage

To quantify the time-related overhead of the different frameworks, I execute the optimization pipeline of each framework 10 times on the LULESH proxy application. Therein, I record the average time taken by the *profiling* step, the *postprocessing* of the profiling data that all frameworks perform, and the *optimization algorithm* itself.

Figure 5.6 shows the results of this experiment. This plot also includes a baseline for running LULESH once with all data placed in DRAM, marking the lowest possible time any profiling-based framework can achieve. In this experiment, ecoHMEM is the fastest framework, followed by ProfDP's latency-aware optimization approach and H2M's initial data placement (IDP-LT). H2M's phase-based optimization (PBDP) and ProfDP's bandwidth-aware approach take by far the longest.

While H2M and ProfDP both take significantly longer than ecoHMEM to optimize LULESH's data placement, the underlying reasons for this are different. H2M's large time requirement is mostly due to its optimization algorithm taking longer, which is a consequence of it solving a 0/1 knapsack exactly. Exactly solving the 0/1 knapsack also causes H2M's phase-based optimization to take significantly longer than IDP-LT, as it solves one knapsack for each phase in the application (cf. Section 4.1.1). ProfDP's heightened time requirement is due to its profiling step requiring two profiling runs, with the second profiling run taking significantly longer in

the bandwidth-aware approach in particular. These results confirm the concerns I mentioned in Section 4.3.2. For real-world applications with their longer execution times, such overheads border on the edge of unusability, especially when considering ProfDP's mediocre optimization results from Section 5.3.

Figure 5.6, however, does not show the time required to set up the workflow for each framework, as I did not measure it exactly when I put together the respective pipelines. Thus, I will only give brief estimates on how much time and effort the initial workflow setup required.

ecoHMEM was the fastest framework to set up since it provides fully documented, ready-to-use shell scripts that only require setting a handful of environment variables. In total, ecoHMEM's setup took me ~20–30 minutes of time. However, my user experience in this regard is likely better than the average user's, since I could use the provided example configuration files as mentioned in Section 5.1. Other systems that do not use DRAM + Optane NVM would require the user to manually discover their system's memory types and their basic performance characteristics (cf. Section 4.2.2), adding to the aforementioned time overhead.

H2M was the second-fastest framework to set up. While it provides shell scripts similar to ecoHMEM, it does not properly document them, leaving the programmer to figure out the over a dozen environment variables contained in them (cf. Sections 4.1.2 and 4.2.2). This was the main time sink in the setup process, which in total took around 1.5–2 hours of time. Yet, this figure again excludes the first step in H2M's workflow, the memory benchmark, as I could directly use H2M's included configuration files for my test system (cf. Section 5.1). However, since H2M's memory benchmark requires no user input and only needs to be run once per system, I would not consider its runtime to be significant overhead. Similarly, the source code modifications required for H2M's phase-based optimization were minimal. The entire process, from applying Intel's VTune to find LULESH's computational phases to recompiling the modified application, only took around 20 minutes.

ProfDP was by far the slowest framework to set up, as I needed to fully reimplement it myself; I will detail the hardships of my reimplementation in Section 5.4.2. Said reimplementation took ~10–15 hours of work, not including debugging and various bug fixes. This work was also more demanding than for the other two frameworks, requiring me to reread the entire ProfDP paper [61] multiple times to ensure faithfulness to ProfDP's original implementation.

5.4.2. Encountered Hardships

In all three frameworks that I tested, the biggest hardships I encountered were **bugs** within the frameworks' tools. Cumulatively, these bugs required ~40 hours of work to fix or find workarounds for. As such, they were severely detrimental in terms of the user experience, which is why I will subsequently detail them for each framework.

H2M's largest source of bugs was its profiling tool, NumaMMA, with three bugs in particular affecting my testing. First, NumaMMA aborted its profiling in 2 % to 4 % of the proxy application tests due to an internal assertion failing. I did

```
compute.cpp:2 @ posix
compute.cpp:3 @ memkind/pmem
```

(a) Human-readable callstacks

```
compute.exe!00401121 @ posix
compute.exe!0040112c @ memkind/pmem
```

(b) Binary addresses

Listing 2: FlexMalloc's different callstack formats

not debug this issue in-depth, since the workaround for it was simply rerunning the profiling. Second, the Ligra proxy application always threw a segmentation fault when profiled with NumaMMA. Debugging revealed this was caused by Ligra using a customized memory allocation routine, which NumaMMA did not correctly interact with. The fix for this bug was to set the undocumented (!) `-c` command-line option in NumaMMA, which has NumaMMA ignore such custom allocation routines. Yet, using `-c` also causes a segmentation fault, but only after outputting *most of* the profiling data. While this is an improvement over not being able to profile the application at all, it still irrecoverably loses some data and yields malformed output files, which I needed to manually fix before passing them to H2M's optimizer.

Lastly, NumaMMA did not work with MPI applications. I originally planned to include two real-world applications in my tests, OpenFOAM [26] and LAMMPS [56], both of which use MPI. Yet, both applications immediately produced a segmentation fault in `MPI_Init` when profiled with NumaMMA, independent of the used MPI implementation. While using the `-c` option avoided these initial segmentation faults, the profiling still did not complete successfully. Specifically, NumaMMA only output a tiny, unusable fraction of the profiling data before it aborted after myriads of out-of-memory errors. Thus, NumaMMA was one of the two reasons I could not use MPI applications in my tests; I will detail the second reason shortly.

ecoHMEM showed bugs in all steps of its workflow. First, its profiler, Extrae, has a race condition in the initialization of the sampling hardware counters. Said race condition caused the profiling to either freeze or outright crash with a segmentation fault in ~10 % of cases, thus necessitating a fix.[3] Further, Extrae's profiling output for the Ligra proxy application caused the optimization scripts to abort. ecoHMEM's scripts expect the profiling output to be sorted by timestamps, which it was not for this specific application. Instead of debugging the underlying issue, I manually sorted the output data to resolve the problem. The optimization scripts themselves also had a bug in their callstack parsing, leading to malformed outputs for the FlexMalloc location files in some cases. I fixed this bug with 10 lines of code changes.

ecoHMEM's three most influential bugs, however, originated from FlexMalloc. First, for its interception of memory allocation calls, FlexMalloc needs to match the human-readable callstacks in the location file (cf. Section 2.3, Listing 1) with the allocation calls in the compiled executable. Yet, this matching process (for which FlexMalloc uses `libbfd`) did not function properly. Due to the complex nature of `libbfd`, I was unable to locate and fix this bug. As a workaround, I used *binary addresses* instead of human-readable callstacks in FlexMalloc's location

[3] As mentioned in Section 5.1, I do not use the newest version of Extrae for my tests. However, this race condition is still present in the newest version of Extrae (4.3.1).

file. For this, I set the `-no-translate-data-addresses` flag in Extrae to output the binary addresses of the allocation calls in the compiled executable, which ecoHMEM consequently used in the location files as exemplified in Listing 2. Using these binary addresses revealed the second bug, which caused FlexMalloc to mistake the absolute addresses in the location file for relative offsets in some circumstances. I fixed this bug by adding 5 lines of code to handle this edge case.

The third FlexMalloc bug pertains not only to ecoHMEM but also to the other two frameworks and my general test setup. As mentioned in Section 5.1, I use H2M's customized FlexMalloc fork and its provided allocators for all frameworks since `memkind` did not recognize the test system's NVM. However, H2M's provided FlexMalloc allocators did not interact properly with MPI applications. As soon as FlexMalloc redirected any allocation to them in an MPI environment, the application immediately crashed with a segmentation fault. Even after extensive debugging (> 15 hours) using memory checkers such as Valgrind [42], I could not identify what exactly causes this issue. Since all three frameworks use FlexMalloc, I thus had to abstain from testing any MPI applications with any framework.

ProfDP was also indirectly affected by all the aforementioned bugs, since my ProfDP reimplementation uses both Extrae and FlexMalloc. However, since I reimplemented ProfDP myself, discussing any bugs that I found and fixed in the optimization algorithm scripts is superfluous since these were of my own wrongdoing.

Apart from these bugs, I also encountered **other hardships** when using these frameworks. However, in comparison to the previously discussed bugs, these were rather minute in nature, which is why I will only mention them briefly.

For H2M, the biggest hardship apart from the bugs was the missing documentation already mentioned in Sections 4.1.2 and 5.4.1. While I was able to deduce the meaning of most environment variables in the provided shell scripts, having no documentation still had some consequences for my tests. For instance, at one point I had to redo my tests for H2M's phase-based optimization in Section 5.3.3 because I forgot to modify one environment variable (`H2M_DUMP_DIR`), without which H2M does not output the necessary phase information its optimizer requires.

With ecoHMEM, I did not experience such hardships, courtesy of its thorough documentation. The only issue I had to resolve apart from bugs was adding support for my test system's PEBS counters to Extrae. As mentioned in Section 2.2, these hardware registers used for the sampling-based profiling are processor-specific, and Extrae did not include support for the used Ice Lake platform out of the box.

ProfDP had the biggest non-bug-related hardship associated with it by requiring a full reimplementation of its workflow. Apart from the issues already discussed in Section 4.3.2, further impediments arose during my reimplementation. For example, since ProfDP requires two profiling runs, some short-lived objects might be profiled in one profiling run but not the other due to the sampling-based nature of the profiling step. Wen et al. [61] do not address how they dealt with such objects. I decided to ignore objects whose allocation was recorded in only one run but kept objects without profiled accesses if their allocation was found in both runs. Also, reimplementing ProfDP requires the programmer to familiarize themselves with at

least one profiling tool to extract the necessary latency data from it. I settled on Extrae since I was already familiar with it from fixing its previously discussed bugs.

Yet, through my reimplementation efforts, I was able to improve ProfDP's user experience over the original workflow described in Section 4.3.2. Specifically, I automated steps 3 to 5 in ProfDP's workflow, i.e., the placement optimization itself and the facilitation of the optimized placements. For the placement optimization, I chose a greedy decision algorithm. I sort the objects by their moving factors in descending order and subsequently iterate through them, placing as many objects with high moving factors into the faster memory type as possible. I settled on this algorithm since I opine that this is the most likely approach a programmer would use when applying the original, user-driven optimization workflow. To then facilitate these optimized placements in an automated fashion, I use FlexMalloc.

6. Discussion

After testing each framework in-depth in the previous chapter, a high-level review of the gathered results is necessary to succinctly compare the three frameworks and establish areas of improvement for them. Thus, in this chapter, I compare and discuss the results for the performance-related tests (Section 6.1) and each framework's user experience (Section 6.2) on a higher abstraction level. Based on these observations, I then discuss how future work could combine the advantages of the three tested frameworks to yield improved frameworks (Section 6.3).

6.1. Performance-Related Results

In terms of the execution time of the made placement decisions, H2M performed the best out of the three frameworks in a supermajority of cases. The main weakness of H2M's optimization algorithm that my test suite identified is an overemphasis on sequential writes. Put into context with the other frameworks' results, however, this is a minor weakness. In the few cases where said overemphasis had a negative effect on H2M's placement decisions, e.g., in the sequential access pattern configurations of my synthetic benchmark, H2M's placement decisions were the second-best; in this concrete example, its placements were only 6 % to 18 % slower than the optimal placements. Thus, H2M's optimization approach overall is the most promising to build upon, requiring only some "fine-tuning" to address these last weaknesses.

H2M's phase-based optimization yielded almost no benefits over IDP-LT, beating IDP-LT in only the 40 % capacity limit test for LULESH. Further, as mentioned in Section 5.3.3, PBDP can perform significantly worse than IDP-LT depending on the chosen phase granularity. Thus, PBDP's benefits over IDP-LT are currently too speculative and too insignificant to justify the additional time the programmer must afford to use said phase-based optimization. Yet, this might change if H2M's authors enact their plans to make H2M's runtime data migration asynchronous [33].

ecoHMEM, across both the synthetic benchmarks and the proxy applications, was consistently the second-best framework. Its biggest impediment performance-wise was the simplicity of its object value metric, only using cache misses with no further "context information". Yet, in cases such as the synthetic benchmark's sequential access pattern configurations, this pure focus on cache misses benefitted ecoHMEM, as it was thus (indirectly) able to distinguish different access patterns. Therefore, ecoHMEM demonstrates that simple composite metrics can be beneficial for identifying specific characteristics but are ultimately insufficient to reflect the inherent complexity of the data placement optimization problem. Further, one main compo-

nent of its optimization algorithm, the bandwidth-aware object classification, proved ineffective. It had no effect in both synthetic benchmark experiments and only altered placements in one proxy application test (LULESH). Thus, while interesting in its approach, it is too situational, requiring a specific iterative application structure with many allocations per callstack to function (cf. Section 4.2.1).

ProfDP's performance was the worst of the tested frameworks over all tests, with no clear trend emerging as to whether its latency- or bandwidth-aware approach is best for the used DRAM + NVM system. However, this does not mean its optimization algorithm is fully without merit; for example, ProfDP was the only framework that consistently identified `p_3` as the performance-critical object in the synthetic benchmark's sequential access pattern configurations (at least in the DRAM profiling experiment). This demonstrates that the latency data obtained from sampling-based profiling can be valuable. However, relying on this latency data *alone* in relative sensitivity metrics has shown too many weaknesses: the missing write latency data causes suboptimal decisions for objects with many writes; the focus on relative metrics causes suboptimal decisions for objects with high baseline access latencies; NVM's hardware sensitivities heavily affect ProfDP's results; latency spikes during profiling skew its placement decisions. In combination, the forenamed points show that latency data can be a valuable addendum for already existing optimization algorithms but should not form the sole basis for such.

Noteworthy for the performance-related results is that in all experiments, the power draw differences between different placement decisions were minimal. Therefore, the framework that optimized the execution time the most automatically optimized the energy consumption the most. This partially contradicts the results of Katsaragakis et al. [29], who report that energy-focused optimization can reduce energy consumption by as much as 40 % over pure performance-focused optimization. Yet, since they only give relative result numbers and do not specify how they measured the energy consumption, I question their measurements based on my results. Still, future work could explore the potential benefits of adding energy-aware components to the frameworks, since minute power draw differences *did* exist. Yet, my results show that this should not be the main focus for improving the frameworks.

6.2. User Experience-Related Results

In terms of the user experience, the most significant problem of all frameworks by far is the myriad of bugs that I encountered in my tests. I focus on these bugs and their detrimental impact on the frameworks' user experience to such an extent because, in most cases, they were not edge cases caused by a specific combination of tools in my tests.[1] For instance, discovering H2M's non-support for MPI applications required no specific setup. Instead, running any (!) MPI application through H2M's workflow reveals the bugs detailed in Section 5.4.2.

[1] Such bugs caused by a specific combination of tools *did* exist. However, I did not discuss those, as they are specific to my test setup and thus irrelevant to the wider discussion of user experience.

However, to put my critique into perspective, I must also consider that all tested frameworks at the current point in time are research software and thus subject to conference deadlines and grant allocations. As such, quality assurance is understandably not a main development focus. Nonetheless, the aforementioned bugs deserve attention due to their wide-ranging consequences detailed in Section 5.4.2; especially H2M's missing support for MPI applications is a major disadvantage for its user experience because of MPI's ubiquity in the HPC domain. Further, the time the programmer must expend to debug or work around these issues is the largest impediment to these frameworks seeing widespread use across the HPC domain.

H2M's and ProfDP's user experience was further affected by their incomplete (or outright missing) documentation and their large time overheads. ProfDP's profiling overheads and the time required to (re-)implement its workflow in particular are so large that I deem them unacceptable for any real-world use case, especially considering the availability of the other two frameworks and ProfDP's mediocre performance results. H2M's optimization time overheads are also significantly larger than ecoHMEM's, on par with ProfDP for its phase-based optimization (cf. Section 5.4.1, Figure 5.6), but are more fixable than ProfDP as I will detail in Section 6.3.

Thus, overall, I rank H2M's user experience higher than ProfDP's, with ecoHMEM providing a better experience than both of them. However, all frameworks have vast room for improvement in terms of their user experience due to the presence of bugs. Fixing these bugs would yield the highest benefit for improving the frameworks' user experience. Thus, this should be the main focus in the frameworks' further development instead of pursuing ever-diminishing gains by fine-tuning the optimization algorithms.

6.3. Improvements for Future Frameworks

For discussing future improvements to the frameworks, I am going to take H2M as a basis since my test results show that its optimization algorithm is the most promising of the tested frameworks. For improving H2M's **optimization algorithm** further, the bandwidth and latency figures it uses are of particular interest.

H2M's memory benchmark generates two bandwidth-latency curves for each of the system's memory types, one for estimating the read latencies and bandwidths and one for estimating the write latencies and bandwidths. From these curves, H2M chooses the bandwidth and latency figures it uses in its optimization rather simply: the latency always is the measured idle latency, and the bandwidth is the bandwidth measured for the number of threads that the application used during the profiling step. This simple choice ignores how the access pattern or other performance-critical access dimensions might skew these metrics, causing the overvaluing of sequential writes explored in Section 5.3.1. To improve this selection process and avoid the aforementioned biases, I contemplate two different approaches.

The first option uses the latency data profiled from the hardware counters, which can provide valuable information, as ProfDP's results demonstrated. Therein, H2M

takes the profiled latencies and compares them to the latency measured by its benchmark for the memory type the application's data was placed in during the profiling step and the number of threads the application used during said profiling (the "expected" latency). For each object, this yields a relative difference between both metrics (e.g., the profiled latency was 1.6× the expected latency). H2M then uses this object-specific difference to scale the bandwidth-latency curves for all *other* memory types, i.e., multiply the latency and bandwidth value of each point on the curve by this relative difference. From these scaled curves, H2M finally selects its bandwidth and latency numbers for this specific object, again based on the number of threads used for the profiling step. Since NumaMMA already profiles the necessary latency data, implementing this option would require relatively little effort.

The second option entails generating more bandwidth-latency curves for different read-write ratios and access patterns, which memory benchmarks such as Mess [17] or Hopscotch [2] can facilitate. Then, H2M determines which bandwidth-latency curve to use for obtaining the bandwidth and latency numbers on a per-object basis via the object's read-write ratio and access pattern. However, this is significantly more effort to implement correctly, since access pattern detection using *sampling-based* profiling data is a non-trivial endeavor. As the results for ecoHMEM and ProfDP show, using the number of cache misses at different cache levels in combination with the profiled access latencies could at least distinguish between sequential and non-sequential access patterns. However, an extensive literature review shows there is no ready-to-use solution for this problem.

Also, as shown by ProfDP's results for the Ligra proxy application, including a mechanism to treat logically related groups of objects as one for the placement decisions might improve said decisions for real-world applications. However, implementing such a mechanism is difficult, likely requiring static program analysis or instrumentation to identify such groups. This would again increase the time taken for the placement optimization. Further, the gain of including such features is uncertain; in my tests, only one out of four proxy applications showed significant gains because of such group effects. Nonetheless, group identification might be an area worth exploring for further (if minute) improvements in optimization efficacy.

Improving H2M's **user experience** is largely down to fixing the bugs identified in Section 5.4.2. However, since implementing such bug fixes is a lengthy process, I propose two easy-to-implement fixes to noticeably improve H2M's user experience in other ways. First, documenting the already existing shell scripts and their environment variables would ease the discussed configuration burdens. Further, while fixing bugs can take a long time, merely providing the end user a list of known issues and incompatibilities would already improve the user experience substantially. Second, to improve H2M's long optimization time, H2M could employ specialized algorithms for solving, e.g., the *temporal* knapsack problem arising from its IDP-LT approach instead of recursively solving knapsacks with general integer linear program solvers. Clautiaux, Detienne, and Guillot [9] have already proposed such algorithms and provide implementations for them, which H2M could use.

7. Conclusion

In this work, I evaluated and compared the performance and user experience of three state-of-the-art data placement optimization frameworks for heterogeneous memory systems: *H2M*, *ecoHMEM*, and *ProfDP*. Using my custom-developed synthetic benchmark, I identified the systemic strengths and weaknesses of each framework's optimization algorithm. Therein, H2M had the least consequential weakness with its overvaluing of sequential writes, making it the best-performing framework overall. ecoHMEM's biggest weakness was its focus on cache misses. While this focus mostly weakened ecoHMEM's performance, it helped ecoHMEM perform better than H2M in some circumstances, making ecoHMEM the second-best framework overall. ProfDP's biggest weakness was its sole reliance on profiled latency data in the form of relative sensitivity metrics. These caused its placement decisions to be inconsistent for write-heavy objects accessed by few threads, leading to it performing the worst out of the three frameworks. Using four proxy applications, I confirmed that all the aforementioned weaknesses affect the frameworks' placement decisions in real-world circumstances in roughly the same magnitude as in my synthetic benchmark tests.

Regarding the frameworks' energy efficiency benefits, my tests showed that the placement decisions optimizing the application performance automatically minimized the energy consumption the most. This is because the differences in power draw between the different frameworks' placement decisions were minimal. As such, H2M's placement optimization was also the best regarding the energy consumption, with ecoHMEM being the second-best framework and ProfDP performing the worst.

In terms of the user experience, i.e., the time and effort the programmer must afford to use each framework, my experiments revealed that all three frameworks have major room for improvement. Especially the myriad of bugs found within each framework severely harms their user experience, as these bugs require large time investments to work around. Apart from these bugs, ecoHMEM had the best user experience of the tested frameworks, as it provided ample documentation for the end user. H2M did not supply such extensive user guidance but at least provided usable scripts along with its source code. Thus, its user experience was the second-best in my tests. ProfDP's user experience was the worst of the three frameworks since its authors no longer provide a ready-to-use implementation of ProfDP's methodology. Thus, any programmer wanting to use ProfDP must fully reimplement its workflow, which I deem infeasible for any real-world use case.

My results demonstrate which improvements future frameworks should make to increase their overall efficacy. Regarding performance-related improvements, my tests showed that latency and cache miss data can be valuable additions to already

existing placement optimization algorithms, especially when employed to facilitate a coarse-grained access pattern detection. Yet, my results also establish that the main area future frameworks should improve upon is their user experience. Improving the frameworks' documentation and fixing the myriad of bugs present within them would yield significantly more benefits for the end user than further improving the frameworks' placement optimization algorithms.

Future work may employ my "symbiotic" evaluation strategy, using a synthetic benchmark in combination with proxy applications, to evaluate more frameworks on different heterogeneous memory systems. Especially covering non-performance-focused placement optimization frameworks with my methodology would provide the detailed insights necessary to advance research on multi-goal placement optimization frameworks. Also, using my methodology to compare the frameworks to cache mode setups, i.e., setups where the heterogeneous memory hardware itself manages the data placement through a cache mechanism, would be valuable. Since cache mode setups implicitly migrate data between memory types at runtime, quantifying the exact performance difference between the two approaches could identify in which scenarios dynamic data migration can provide performance benefits for the frameworks. Yet, to yield a fair comparison, such tests necessitate restricting the capacity of the individual memory types in cache mode. This is difficult to achieve, and it is the reason why I did not undertake those comparisons myself.

A. Baseline Results for the Synthetic Benchmark

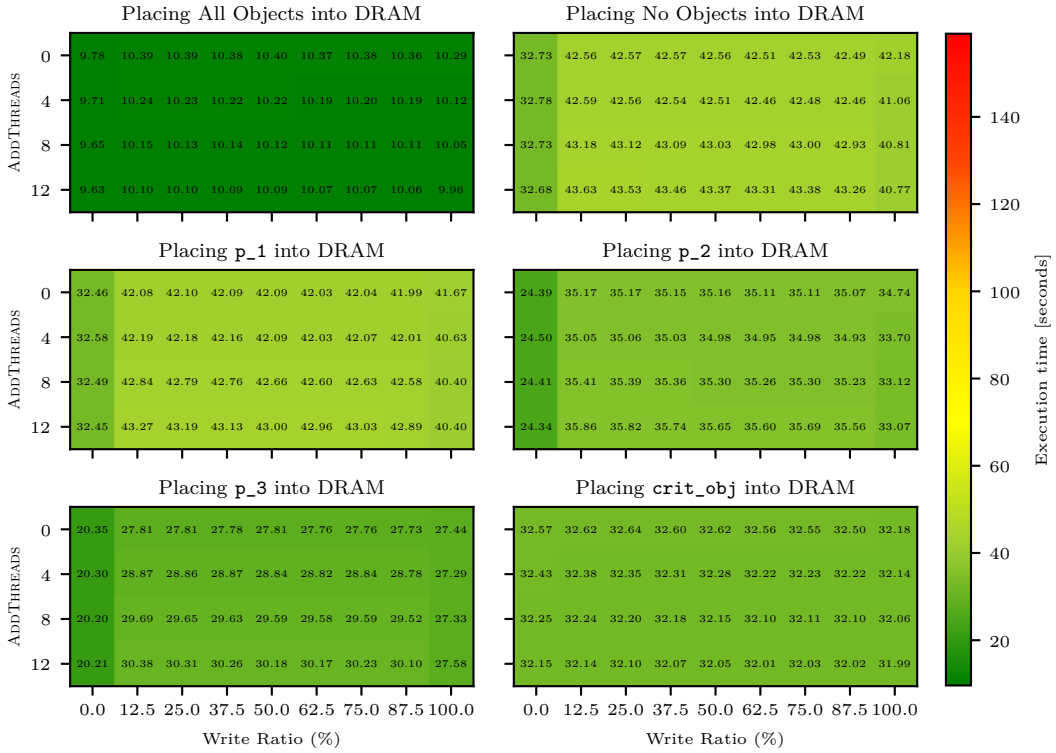
A.1. Execution Time

Figure A.1 shows the execution time for all possible combinations of benchmark configurations and objects placed into DRAM. Its structure is similar to Figures 5.3 and 5.4 in that each subfigure shows the results for the configurations accessing `crit_obj` with a specific access pattern. Each of the bottom four heatmaps in each subfigure then shows the execution time for these configurations when placing the object named in the title of each heatmap into DRAM. Each subfigure further provides two reference heatmaps at the top: one where all four objects are placed into DRAM and one where all four objects are placed into NVM. These establish best- and worst-case baselines for the execution times.

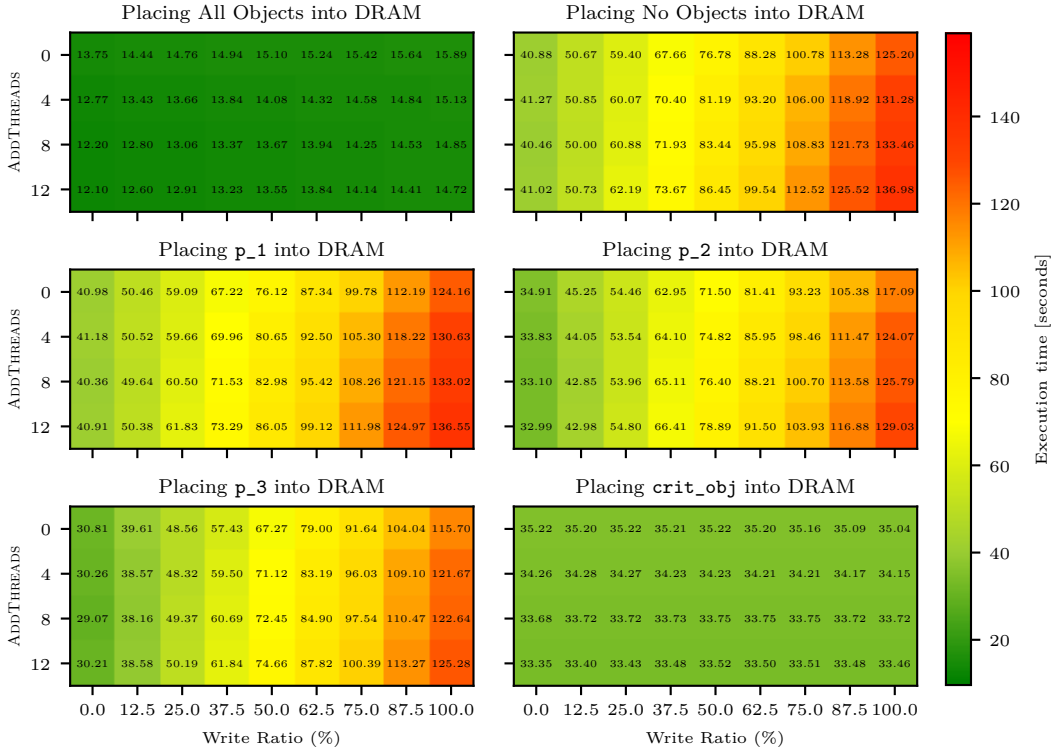
For the benchmark configurations accessing `crit_obj` **sequentially**, Figure A.1 (a) confirms the claim from Section 5.2.1 that placing `p_3` into DRAM is always the optimal decision when placing exactly one object into DRAM. As mentioned in Section 5.2.1, this holds true even when the critical object is written to, corroborating the findings of Yang et al. [63]. Yet, even sequential writes to NVM still have a significant performance penalty associated with them; when `crit_obj` is *not* placed into DRAM, the execution time increases by 10 seconds when switching from a 0 % to a 12.5 % write ratio.

However, increasing the write ratio beyond 12.5 % does not increase the execution time further, even when `crit_obj` is placed into NVM. The reason behind this is that the Optane NVM in the test system conducts its writes in 256 byte chunks, regardless of how many bytes within this chunk were actually modified [45, 63]. Thus, more writes to the same 256 byte region do not cause additional performance degradation. Interestingly, when further increasing the write ratio from 87.5 % to 100 %, the execution time actually *de*creases in a small but measurable capacity. The cause for this behavior is the access-coalescing nature of the read and write pending queues briefly mentioned in Section 2.1.2. In an all-write scenario, the write pending queues can coalesce all sequential (!) writes to a 256 byte region into one singular operation with no "competing" read operation pending in the read queues, thus reducing the overall number of slow NVM accesses.

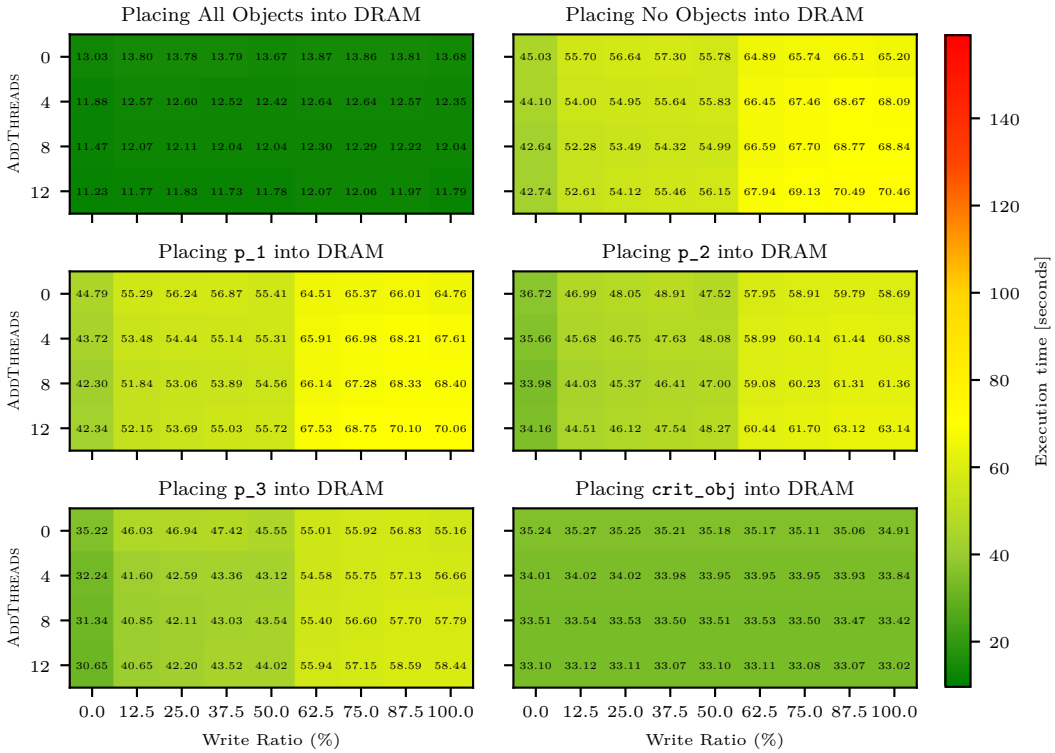
The access-combining nature of NVM's access queues is also the reason why increasing thread-level contention causes increasing execution times only when `crit_obj` is both read from and written to (i.e., 0 % < write ratio < 100 %). When the critical



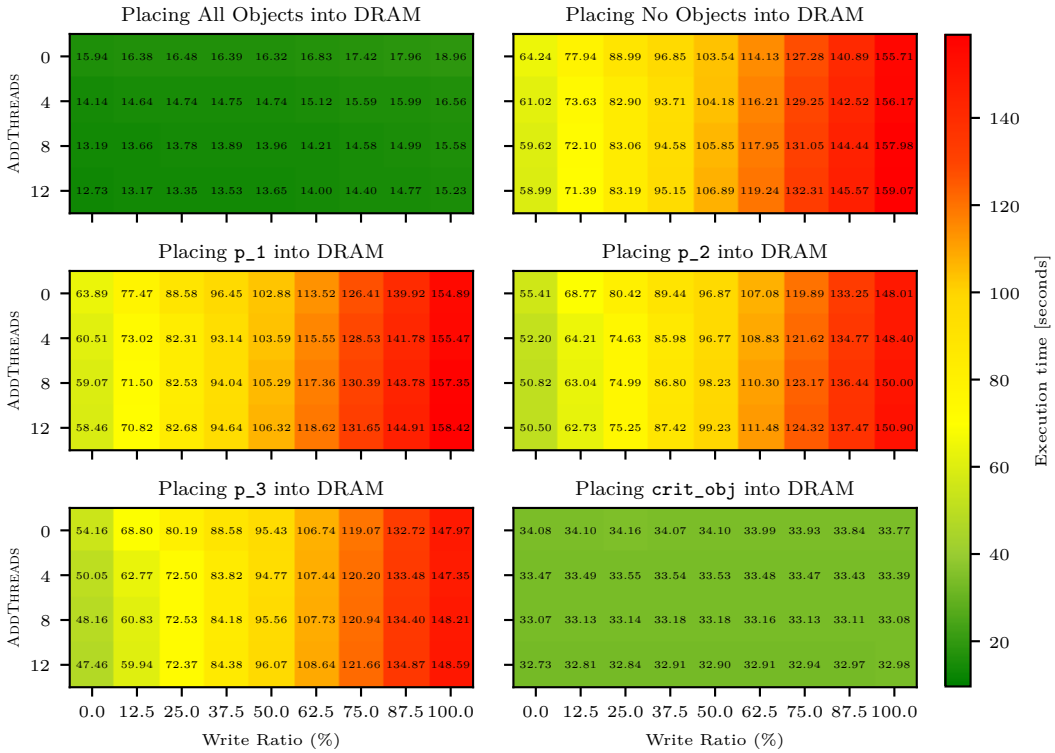
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.1.: Execution times when placing different objects into DRAM

object is only read from or written to, NVM's access-coalescing queues can combine all sequential read or write accesses into one operation, with no competing write or read operation occurring. Thus, execution time *de*creases alongside AddThreads in the configurations where `crit_obj` is only read or written, irrespective of whether it is placed into DRAM or not. Yet, if `crit_obj` is both read and written, execution times increase alongside AddThreads if `crit_obj` is not placed into DRAM. This is because the mixed reads and writes cannot be combined into one operation by NVM's access queues, leading to increased contention on NVM.

In contrast, when `crit_obj` is accessed in a **strided** fashion, Figure A.1 (b) shows that increasing the thread-level contention on the critical object always increases the benchmark's execution time if `crit_obj` is written to and not placed into DRAM. Further, increasing the write ratio beyond 12.5 % now monotonically increases execution times. Both results are the product of the write pending queues being unable to sufficiently coalesce the critical object's strided writes to hide NVM's worse write performance. Thus, placing `crit_obj` into DRAM is optimal if it is written to.

Yet, if the critical object is only read, placing `p_3` into DRAM yields the lowest execution times. This shows that `p_3`'s tiled access pattern is more detrimental to NVM's performance than `crit_obj`'s strided access pattern.

Further noteworthy in this set of results is that the execution times for the all-DRAM baseline increased over its sequential access pattern counterpart (top-left heatmap of Figure A.1 (a)) by as much as 54 %. This does not contradict the "access pattern invariance" I attested for DRAM in Section 2.1.1. Rather, the strided access pattern causes significantly more cache misses than a sequential access pattern (LLC hit rates decrease from 99 % to 50 %). Thus, DRAM's full latency is incurred more often for the strided access pattern, which yields longer execution times.

For the **tiled** access pattern configurations, Figure A.1 (c) most notably shows that the execution time does not increase monotonically when placing `crit_obj` into NVM and increasing its write ratio. Rather, it rises in two distinct steps: when transitioning from a 0 % to a 12.5 % write ratio and when transitioning from a 50 % to a 62.5 % write ratio. This is caused by a combination of the access-combining write pending queues and my implementation of the synthetic benchmark. The write pending queues can combine the writes to each four-element-wide tile (cf. Section 5.2.1, Figure 5.1 (c)), but are unable to combine writes across tiles. When the write ratio is between 12.5 % and 50 %, my synthetic benchmark only writes to the first tile, meaning all writes can be combined into one operation. However, once the write ratio increases above 50 %, the benchmark writes to both tiles, leading to uncombinable writes and thus longer execution times.

Still, increasing the write ratio on `crit_obj` when it is placed in NVM does not lead to the same drastic execution time increases as in the strided access pattern configurations. Further, increasing the thread-level contention on the critical object also does not impact execution time as much as in the mentioned strided access pattern results. Both of these results corroborate Izraelevitz et al.'s [24] findings that Optane's access-combining write pending queues help decrease access contention significantly.

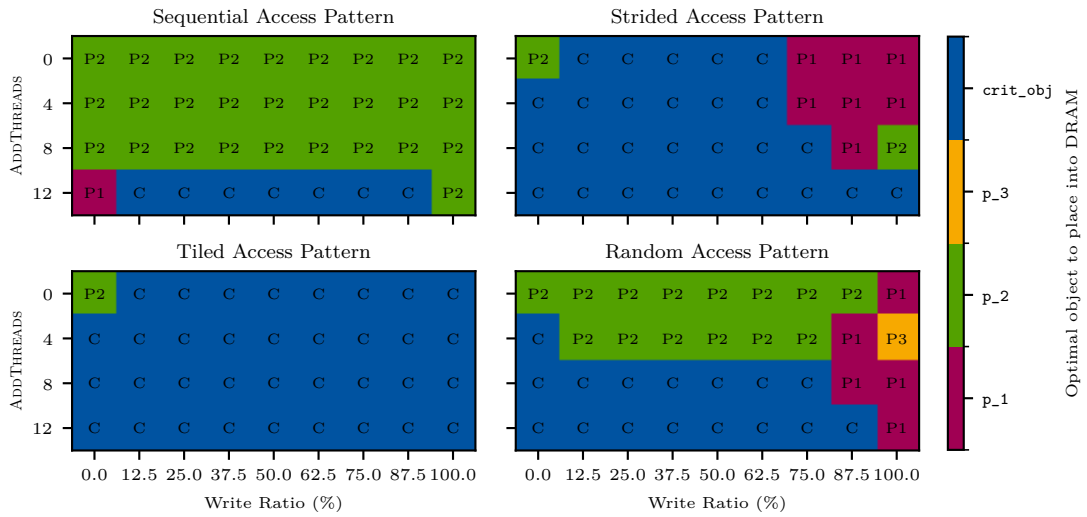


Figure A.2.: Optimal object to place into DRAM to minimize the combined power draw of the `Package` and `DRAM` RAPL domains

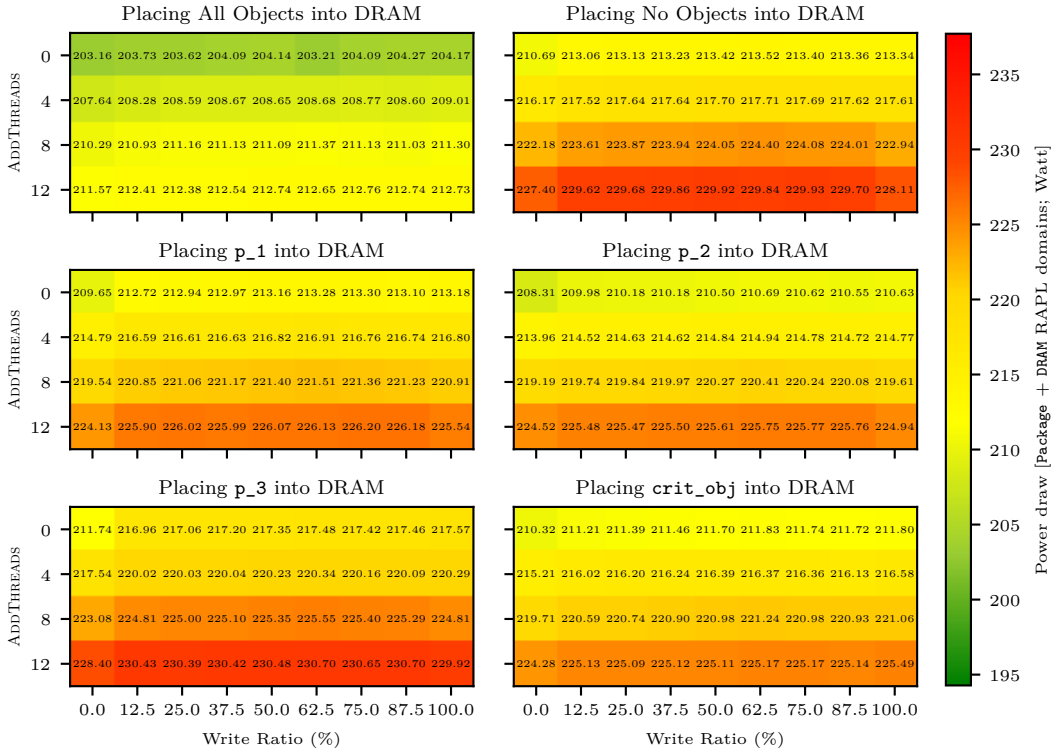
Interestingly, when `crit_obj` is only read, placing `p_3` into DRAM yields execution times up to 2.5 seconds faster than placing `crit_obj` into DRAM. Since this gap only forms when increasing AddThreads, this indicates that the tiled access pattern is unable to saturate NVM's bandwidth. Thus, placing `crit_obj` into *NVM* and accessing it with more threads uses more of NVM's limited bandwidth than if `p_3` is placed into NVM. Since NVM's bandwidth is the bottleneck slowing down the synthetic benchmark's execution time, placing `crit_obj` into NVM therefore yields the lowest execution times.

When `crit_obj` is accessed **randomly**, Figure A.1 (d) shows that placing `crit_obj` into DRAM over `p_3` is the optimal decision even for a write ratio of 0 %. These results corroborate the results of Yang et al. [63] and Izraelevitz et al. [24] that fully random accesses constitute the worst possible access pattern for NVM. Identical to the strided access pattern results, execution time monotonically increases when the critical object is placed in NVM and its write ratio is increased, since the write pending queues cannot combine the random accesses.

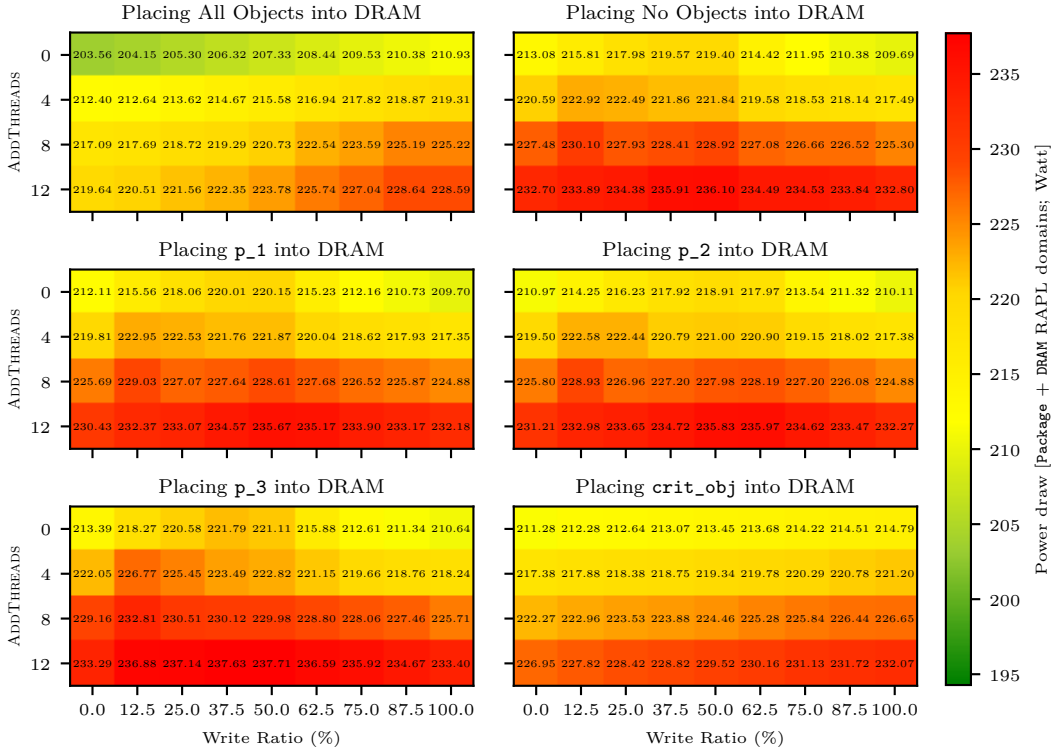
A.2. Power Draw

When considering the system's power draw, results are more mixed than the timing results discussed previously in Section A.1. Figure A.2 shows the optimal object to place into DRAM to minimize the system's *total* power draw, in the same format as Figure 5.2 in Section 5.2.1. Here, no clear pattern emerges as to what object should be placed into DRAM to minimize the power draw across the different configurations.

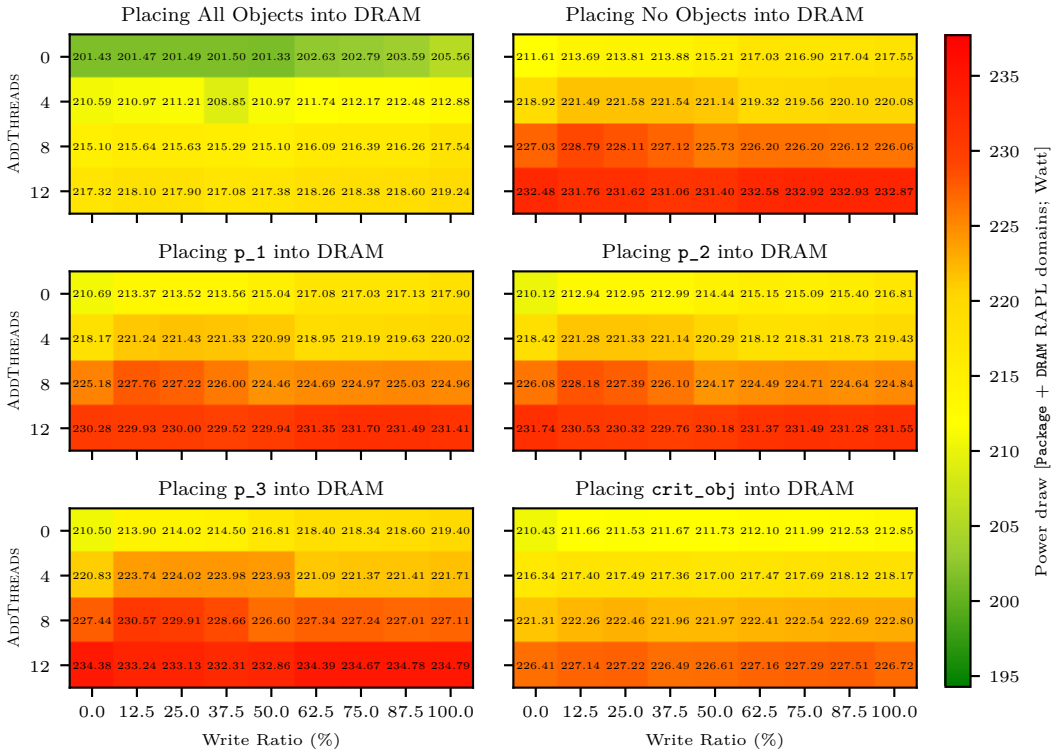
Figure A.3 explains the reason behind this lack of pattern. It shows the power draw for each configuration and placement decision in the same format as Figure A.1. As the figure shows, the differences in power draw between the different placement



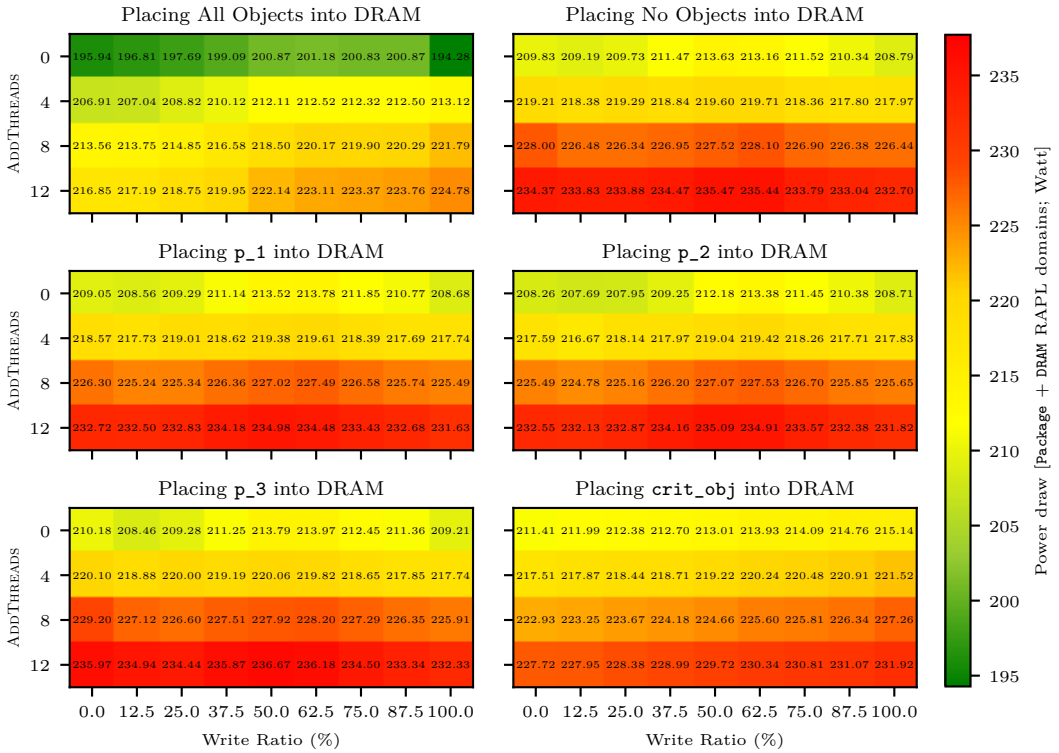
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.3.: Power draw (from the `Package` and `DRAM` RAPL domains) when placing different objects into DRAM

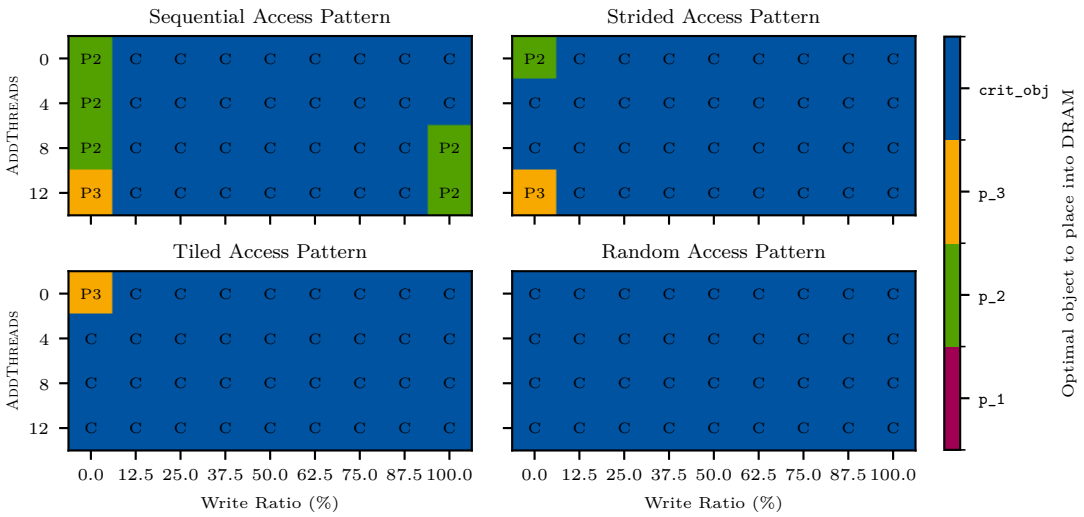


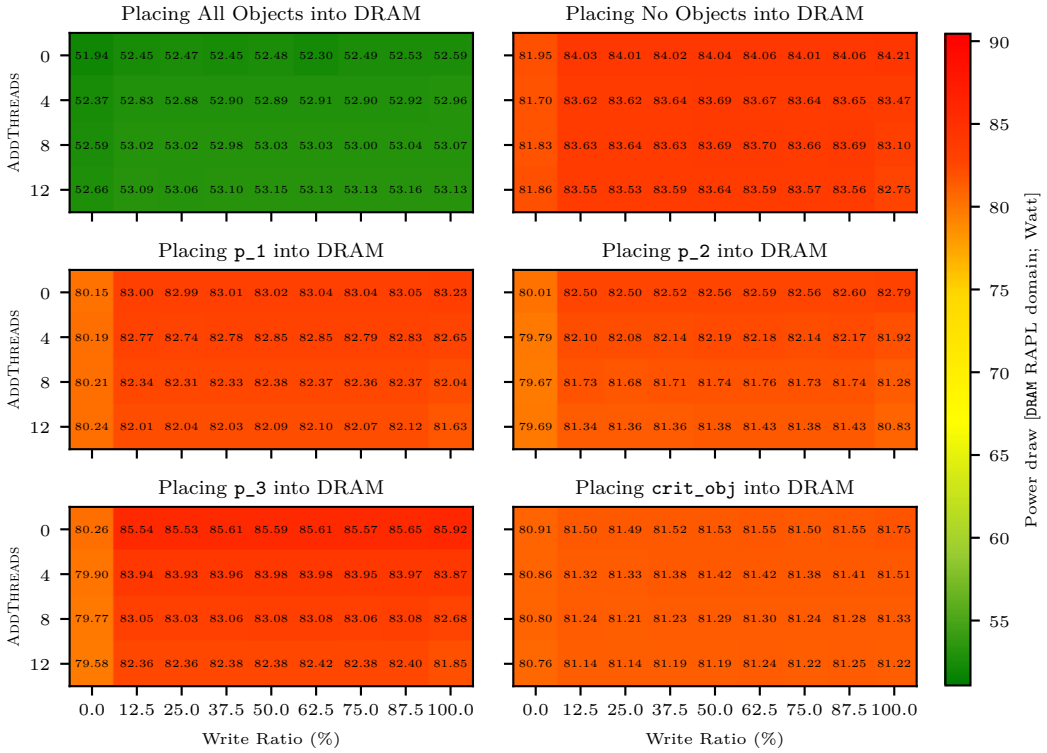
Figure A.4.: Optimal object to place into DRAM to minimize the power draw of the `DRAM` RAPL domain

decisions are minimal for each configuration. The only pattern emerging is that placing `p_3` into DRAM consistently yields the highest power draw, with all other placement decisions being within 2 to 3 watts of each other. Further, placing all objects into DRAM always yields the lowest (!) power draw across all benchmark configurations. Both observations are highly interesting in their own regard. However, explaining the reasons behind them requires breaking down the system's power draw into the individual RAPL domains, which I will consequently do.

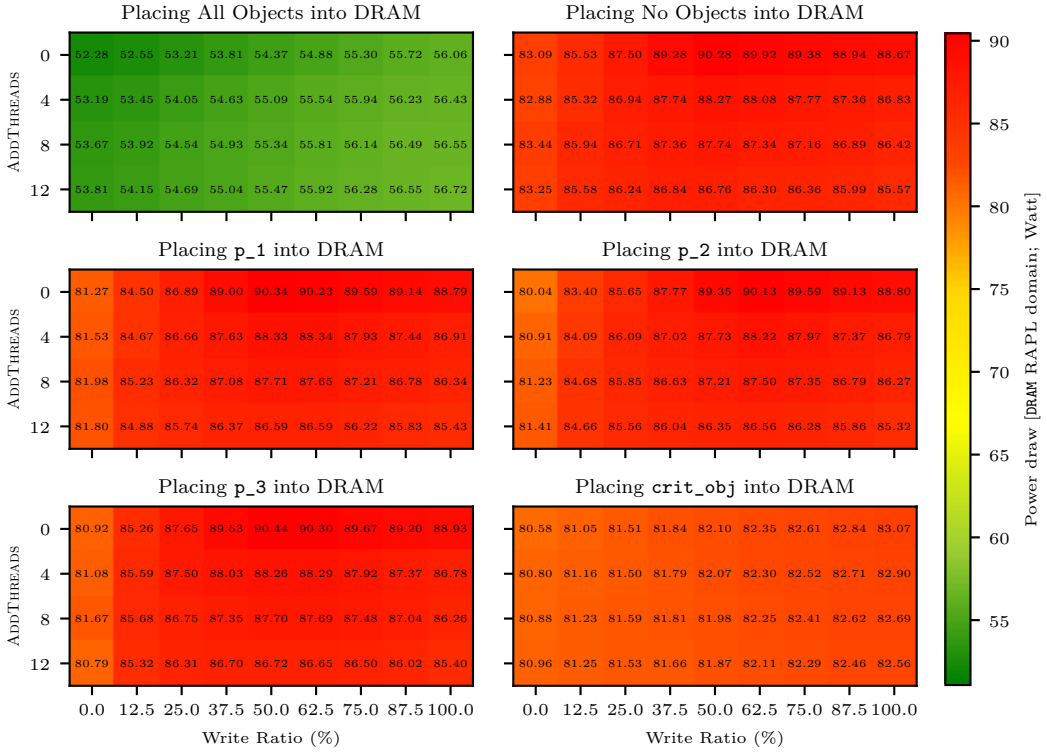
Figure A.4 shows the optimal object to place into DRAM to minimize the power draw of the `DRAM` RAPL domain on its own. As a reminder from Section 2.2, note that the `DRAM` domain measures the power draw of *all* attached memory, not just the DRAM modules. As the figure demonstrates, placing `crit_obj` into DRAM when it is written to is almost always the optimal decision to minimize memory power draw. This is unsurprising, since writing to NVM requires significantly more power than reading from it as explored in Section 2.1.2. In the configurations where `crit_obj` is only read, no clear pattern emerges.

However, when considering the detailed power draw results in Figure A.5, one very intricate pattern emerges. Placing all objects into DRAM consistently yields a ~30 watt lower (!) memory power draw than any other placement decision. Further, these other placement decisions are all within 5 watts of each other. This latter aspect explains the lack of patterns in the total power draw results discussed earlier: If all placement decisions draw similar amounts of memory power, the system's total power draw is mostly determined by the CPU's power draw (i.e., the `Package` domain). Said CPU power draw is subject to more complex factors and interactions (as I will detail shortly), causing the lack of pattern in the total power draw.

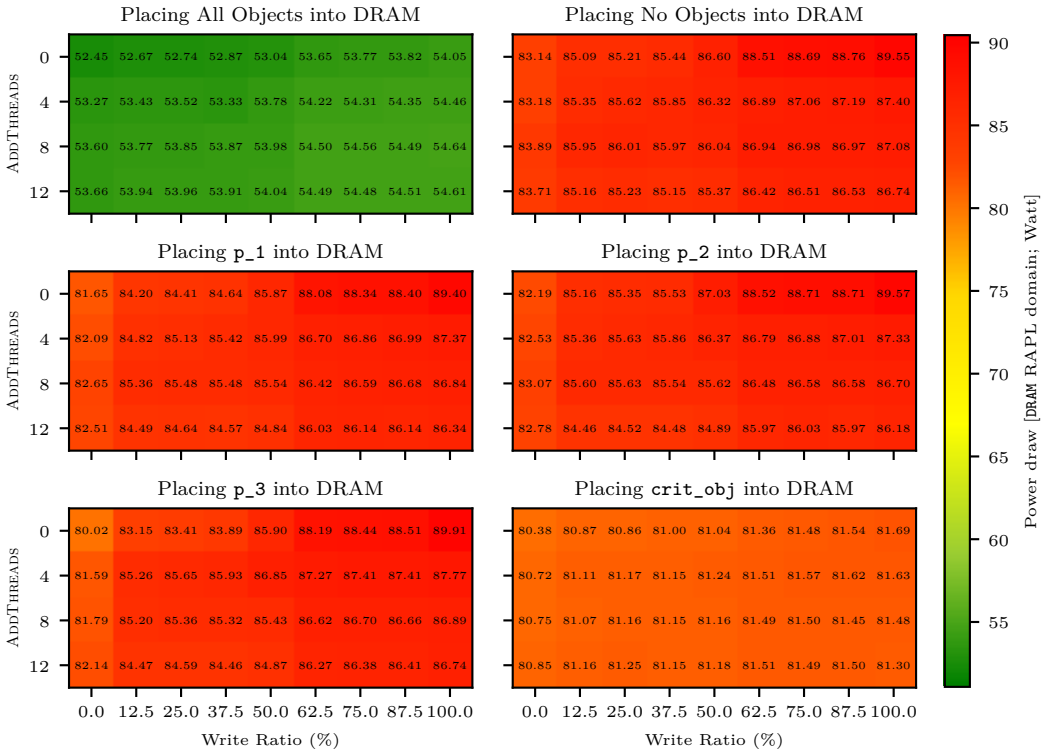
That an all-DRAM placement draws significantly less power than placing any object(s) into NVM corroborates the findings of Klinkenberg et al. [32], Alt et al. [4],



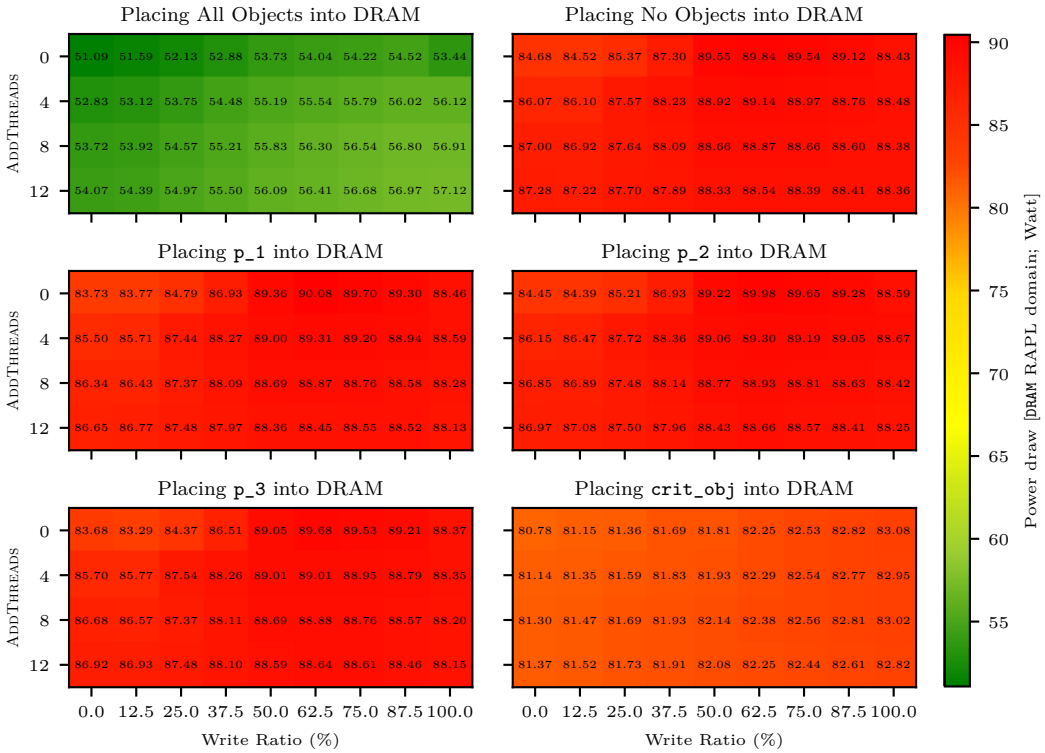
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.5.: Power draw (from the `DRAM` RAPL domain) when placing different objects into DRAM

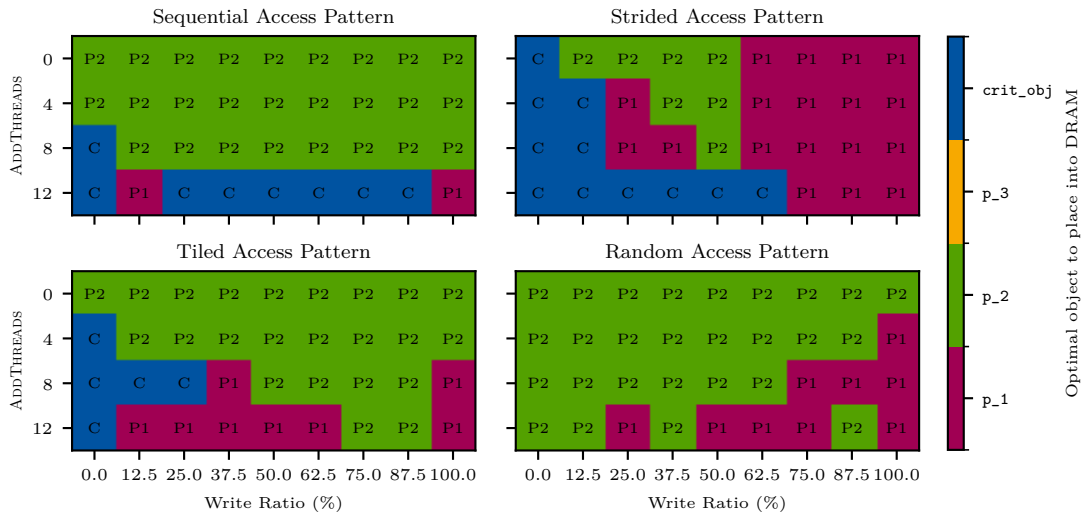
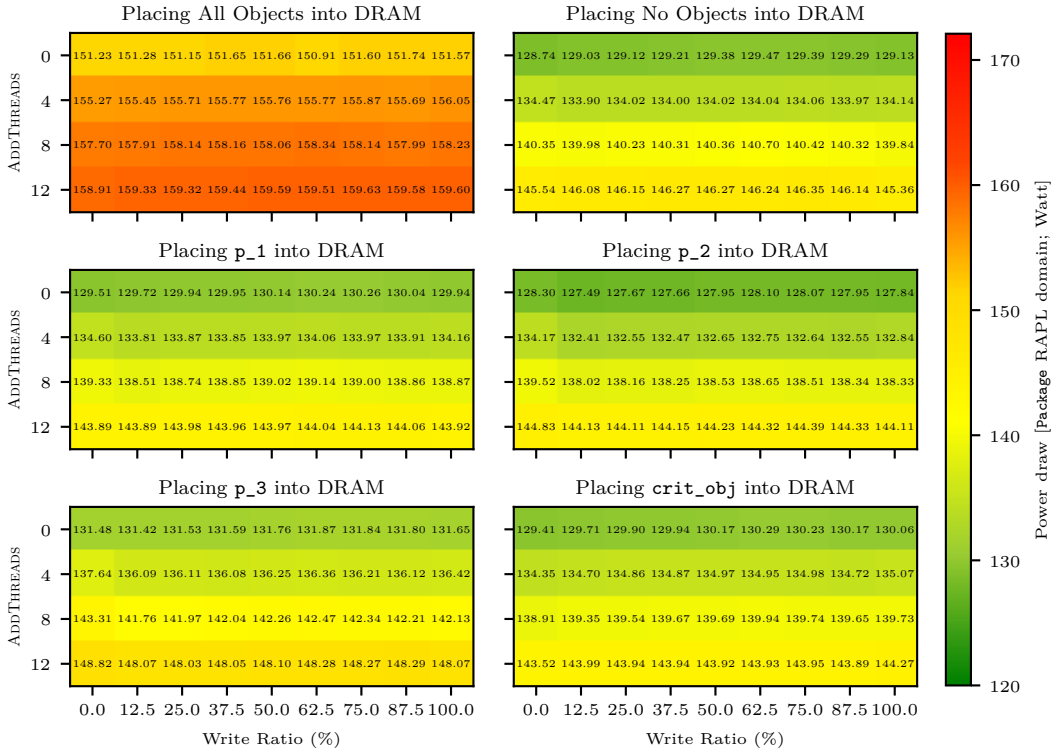


Figure A.6.: Optimal object to place into DRAM to minimize the power draw of the `Package` RAPL domain

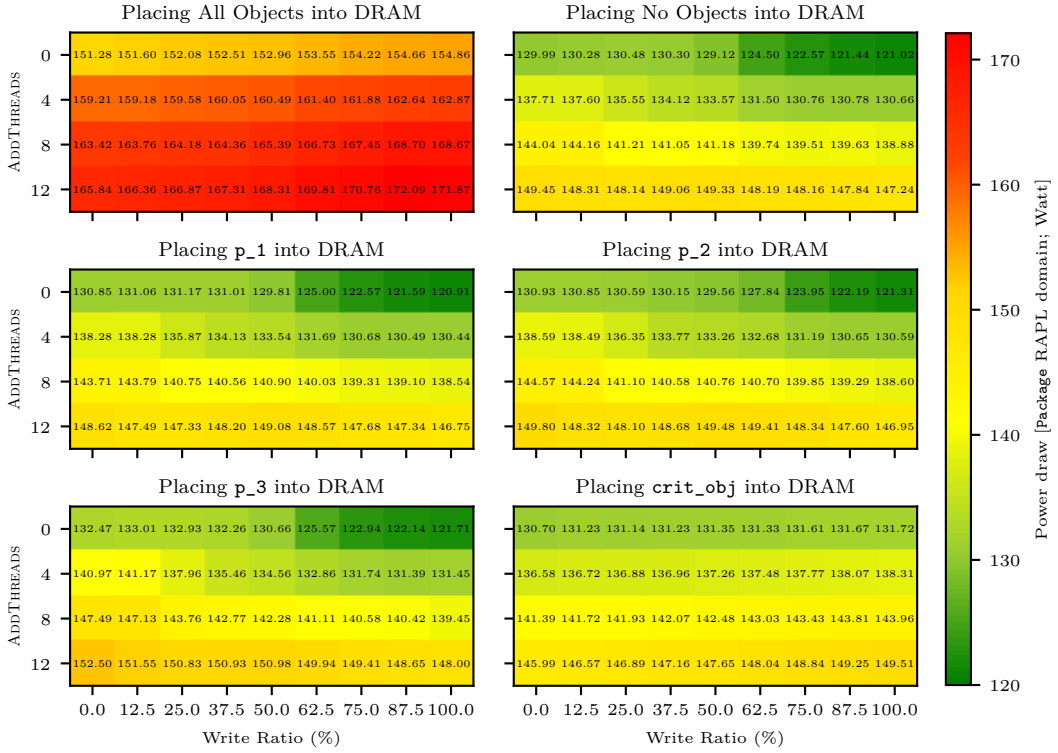
and Preisner et al. [48]. At first, these results might appear counterintuitive considering that NVM has lower power demands than DRAM as detailed in Section 2.1. However, they are consistent with the power draw characteristics in Section 2.1 when analyzing DRAM's power draw in more detail. Most of DRAM's high power demand is due to it needing constant refreshing to hold its data [45]. This *static* power draw is always the same, irrespective of how much DRAM is accessed. Thus, by placing objects into NVM, DRAM's power draw reduces only slightly, while NVM now draws power because it is accessed. Further, actively using NVM and DRAM at the same time may require additional measures to ensure cache coherency [45], thus increasing power draw. Also, Alt et al.'s experiments show that RAPL overestimates NVM's power draw substantially more than DRAM's, skewing results in favor of the all-DRAM placement [4]. In combination, these factors explain the heightened memory power draw when placing any object(s) into NVM.

Reviewing the power draw results for the `Package` domain, Figure A.6 shows that placing `p_1` or `p_2` into DRAM yields the lowest CPU power draw in most configurations. This is because their sequential and strided accesses are the most performance-friendly for NVM, taking the shortest time to complete. Therefore, placing `p_1` or `p_2` into *DRAM* instead of the other objects (whose accesses take significantly longer to complete on NVM) maximizes the CPU's wait times, yielding lower CPU utilization and power draw. This also explains why placing `p_1` or `p_2` into DRAM minimizes the system's total power draw in some configurations as previously depicted in Figure A.2. The differences in CPU power draw outweigh the minute differences in memory power draw in these configurations, leading to the aforementioned results for the system's total power draw.

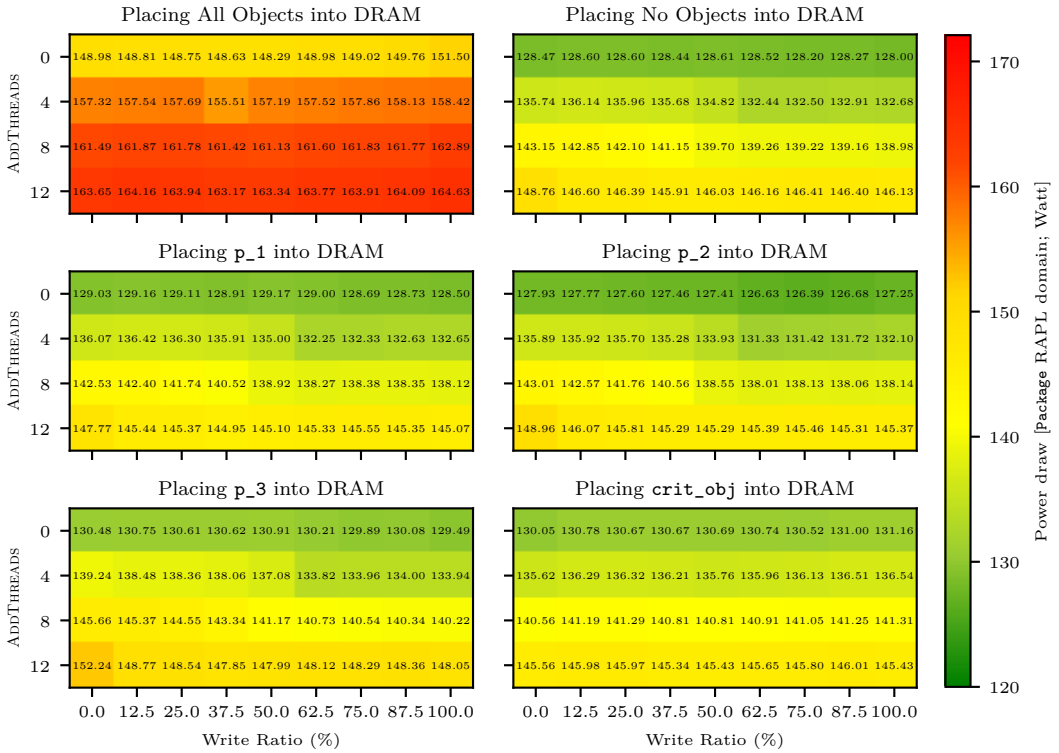
The same wait time argument also explains the detailed power draw results depicted in Figure A.7. The all-DRAM placement has a 15 to 20 watt higher power



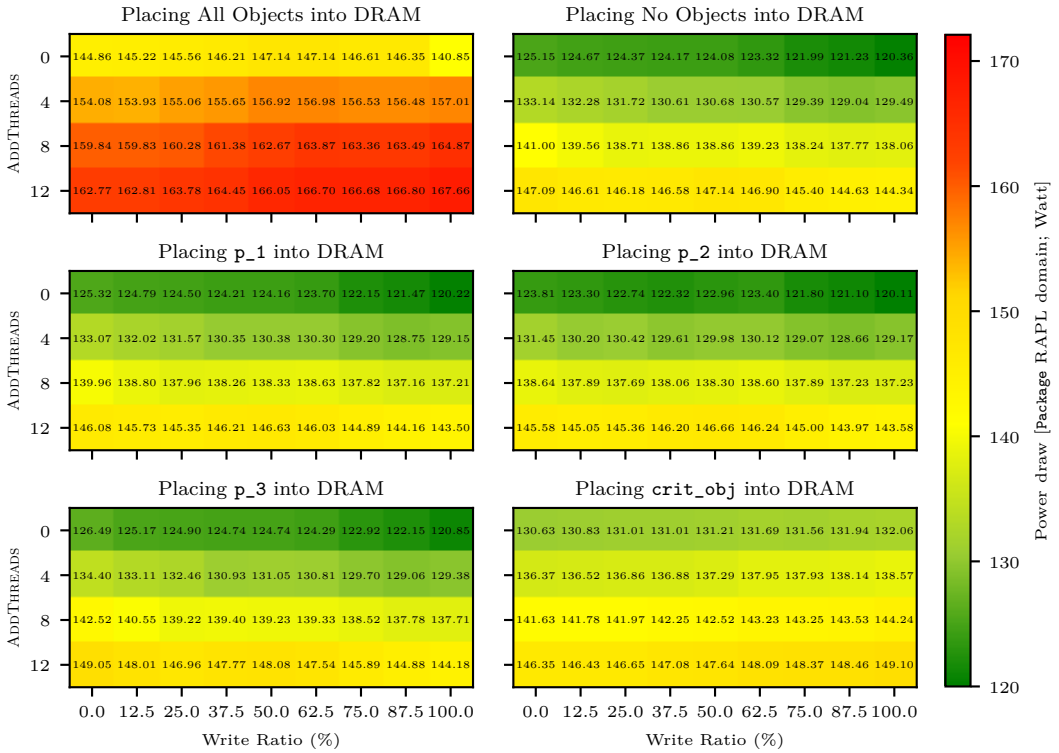
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.7.: Power draw (from the `Package` RAPL domain) when placing different objects into DRAM

draw than all other data placements because DRAM's higher bandwidth and lower latency in comparison to NVM minimize the idle CPU time. Yet, the 20 watt increase in CPU power draw is insufficient to offset the 30 watt lower memory power for the all-DRAM placement. Thus, when considering the system's total power draw in Figure A.3, the all-DRAM placement yields the lowest power draw.

Further, the wait time argument also explains why the power draw decreases when placing `crit_obj` into NVM and increasing its write ratio. Since writes take significantly longer to complete than reads on NVM (as explained in Section 2.1), not placing the critical object into DRAM when it is written to causes higher wait times for the CPU. This decrease is also significant enough to overcome the 5 watt difference between the memory power draw of the different placement options, explaining why not placing the critical object into DRAM minimizes the system's total power draw in mostly write-intensive configurations.

As an aside, the power figures for the `Package` RAPL domain shown in Figure A.7 also serve as a sanity check for the power measurements, since increasing AddThreads consistently increases power draw. This exactly matches the expected behavior of more active threads increasing CPU power draw.

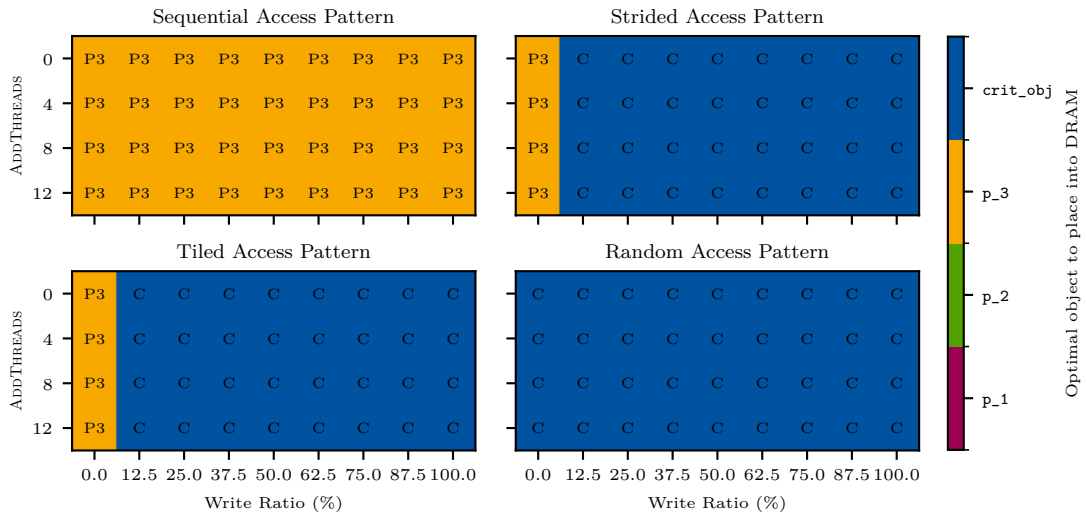


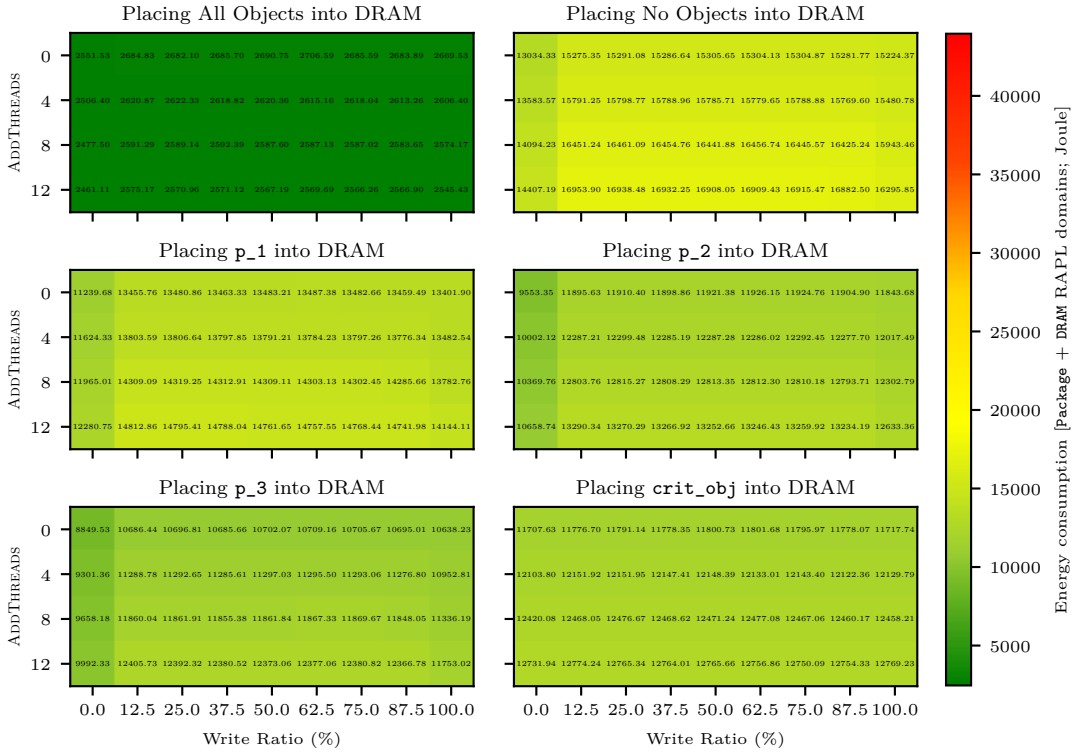
Figure A.8.: Optimal object to place into DRAM to minimize the combined energy consumption of the `Package` and `DRAM` RAPL domains

A.3. Energy Consumption

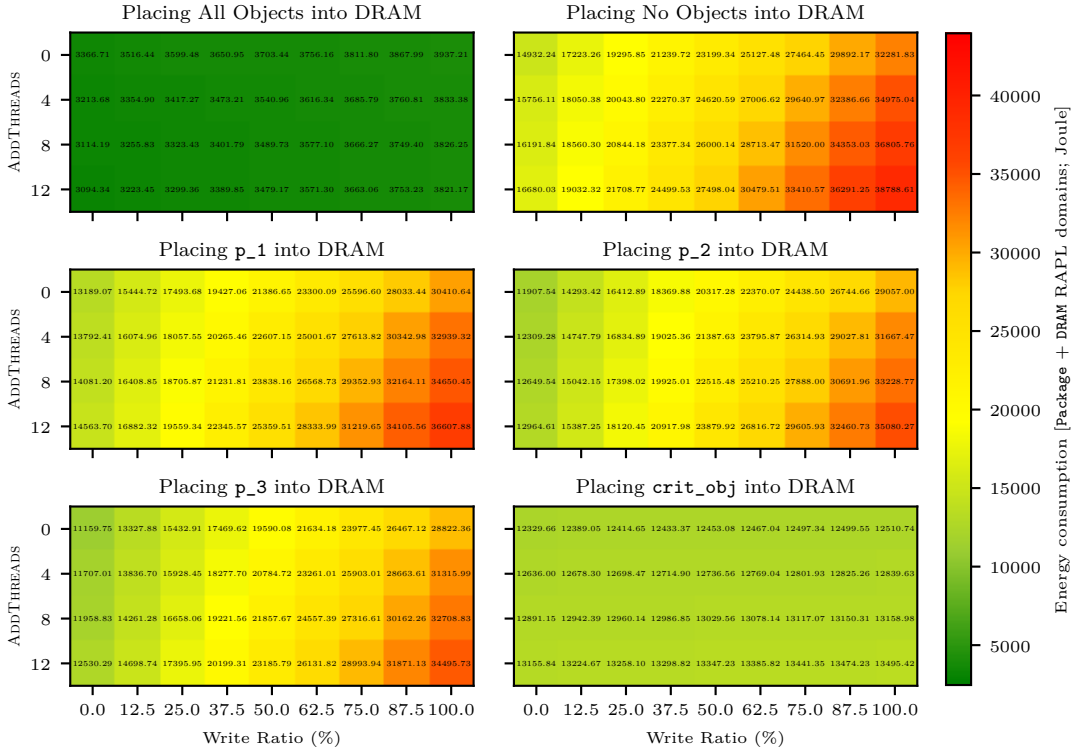
The results for the energy consumption return to being similar to the results for the execution time. As Figure A.8 shows, the optimal placement decisions to minimize the system's total energy consumption are identical to the placement decisions minimizing the benchmark's execution time (cf. Section 5.2.1, Figure 5.2). In light of the previously discussed power draw results, this is to be expected since the differences in power draw were minimal between different placement decisions. Thus, the energy consumption (which equals Power × Execution Time) is mostly dictated by the benchmark's execution time.

For the same reason, the energy consumption for the `DRAM` RAPL domain also follows the same overall pattern as the execution time results, as Figure A.11 demonstrates. Thus, the optimal decisions to minimize the memory energy consumption are also identical to the execution time decisions, as shown in Figure A.10. Yet, the relative difference between the all-DRAM placement and the other placement options is significantly higher for the `DRAM` RAPL domain than for the total system. This is because the all-DRAM placement yields significantly lower memory power draw figures compared to other placements but has a similar total power draw.

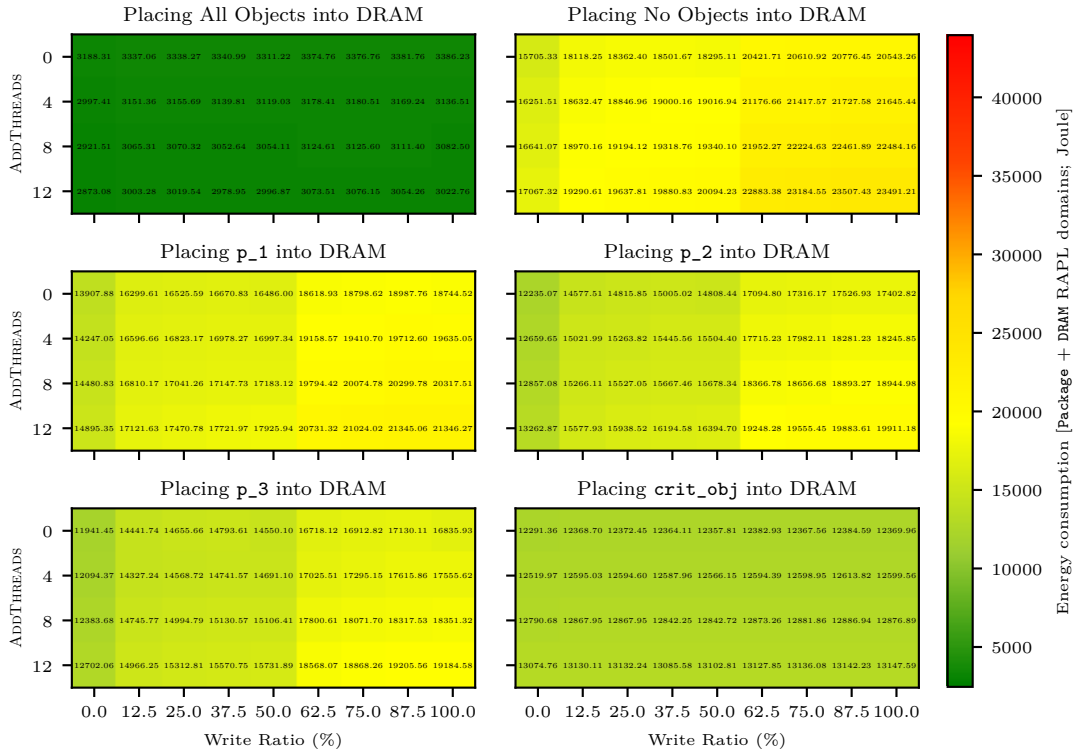
The execution time also dictates the energy consumption of the `Package` RAPL domain, as Figure A.13 shows. Thus, the placement decisions minimizing CPU power consumption are again the same as the optimal execution time placements, as Figure A.12 depicts. Here, the only notable deviation from the execution time results is the heightened impact of the thread-level contention. Increasing AddThreads significantly increases the CPU's energy consumption since more active CPU threads increase the CPU's power draw.



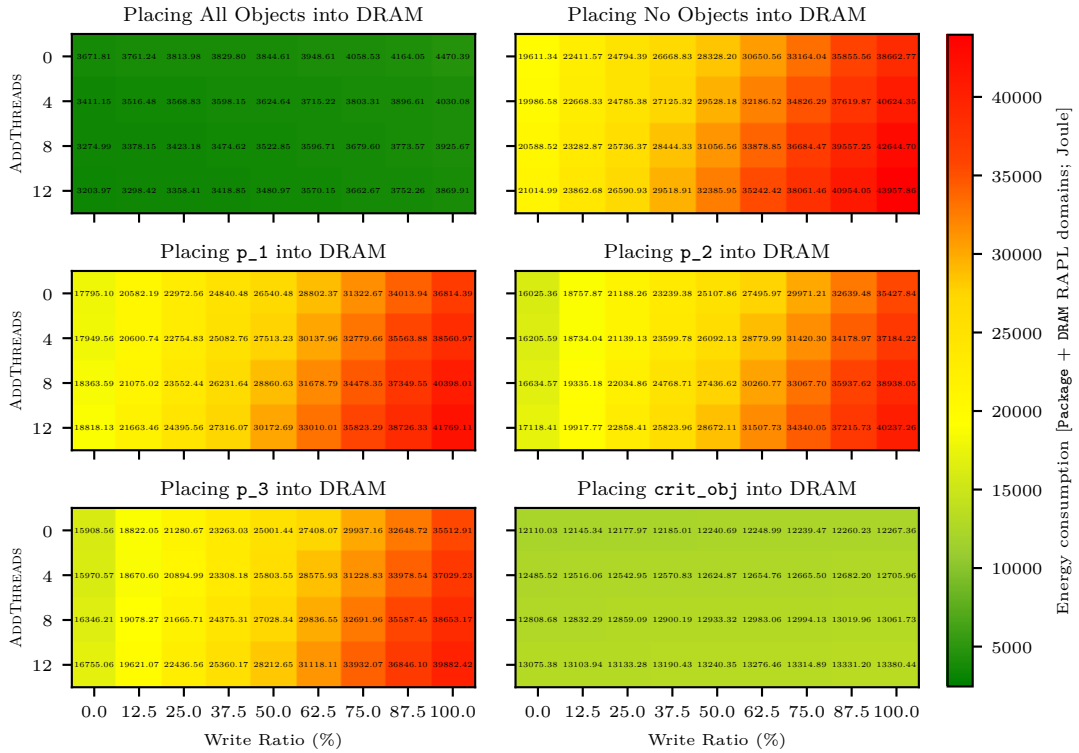
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.9.: Energy consumption (from the `Package` and `DRAM` RAPL domains) when placing different objects into DRAM

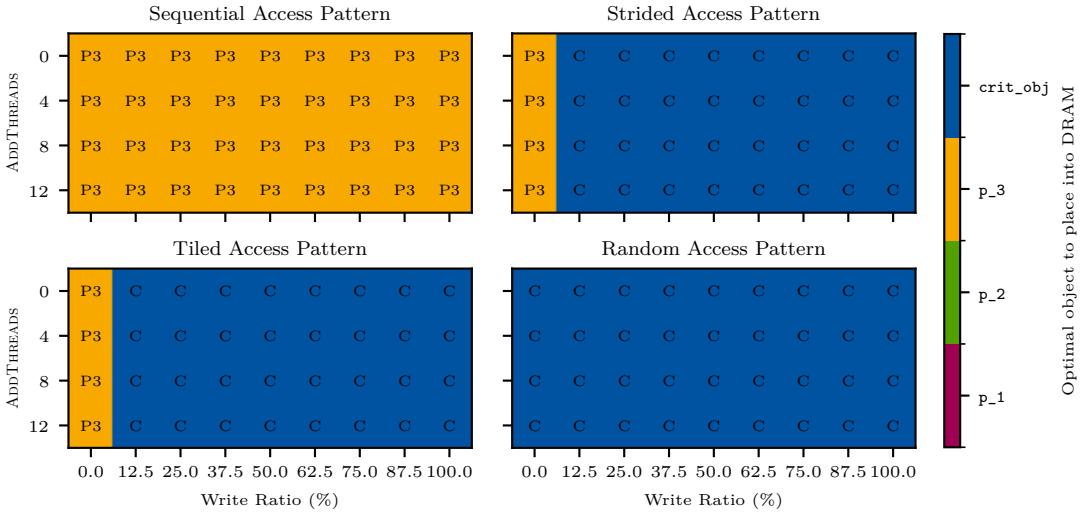
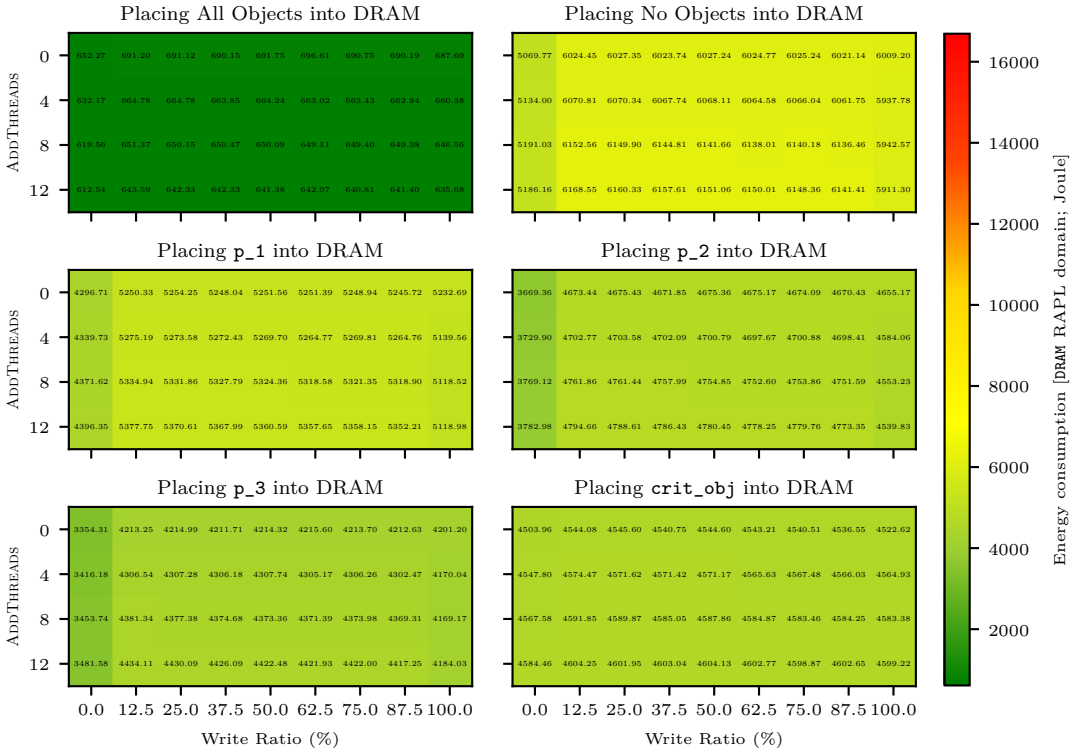
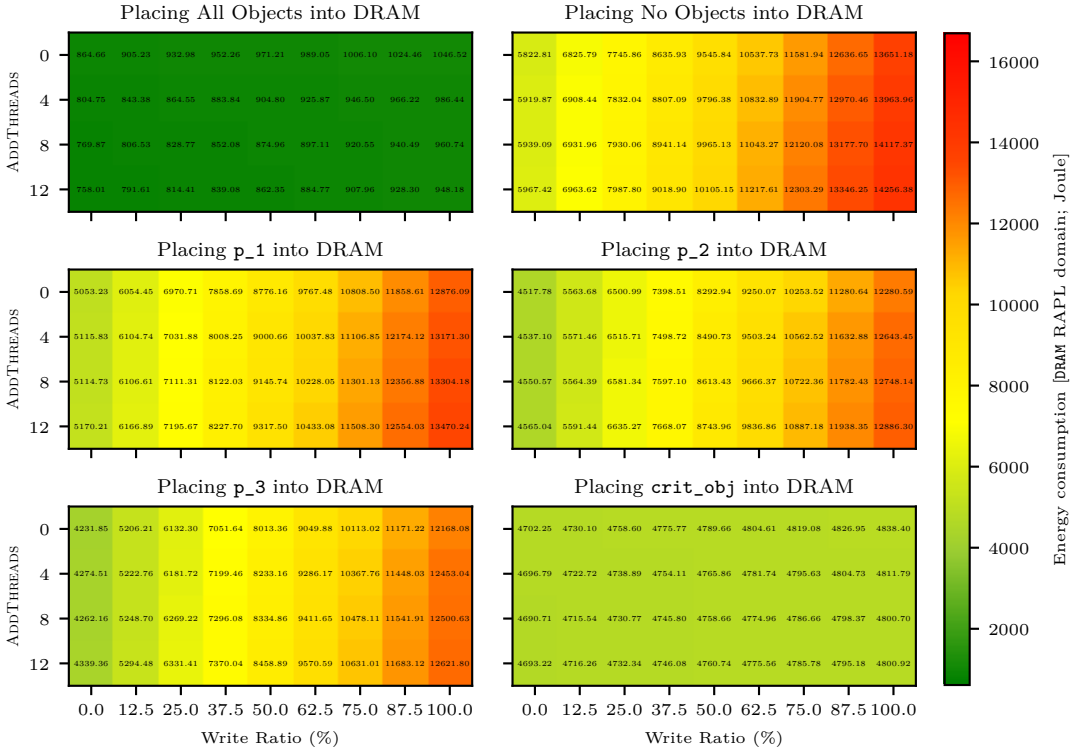


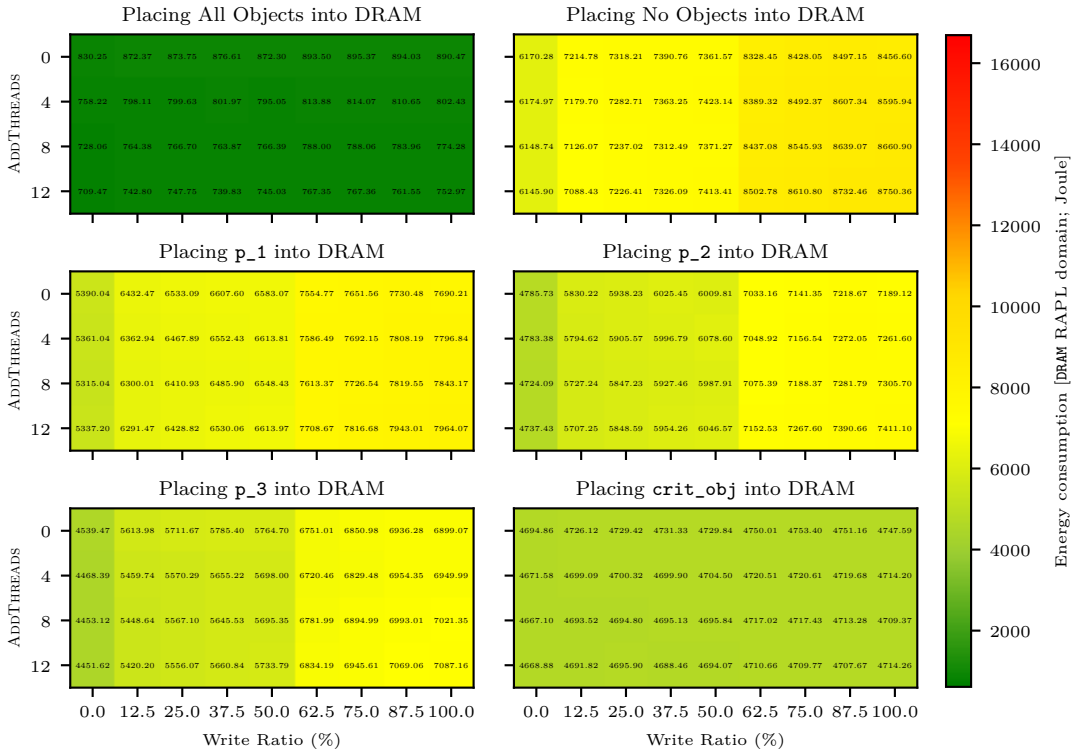
Figure A.10.: Optimal object to place into DRAM to minimize the energy consumption of the `DRAM` RAPL domain



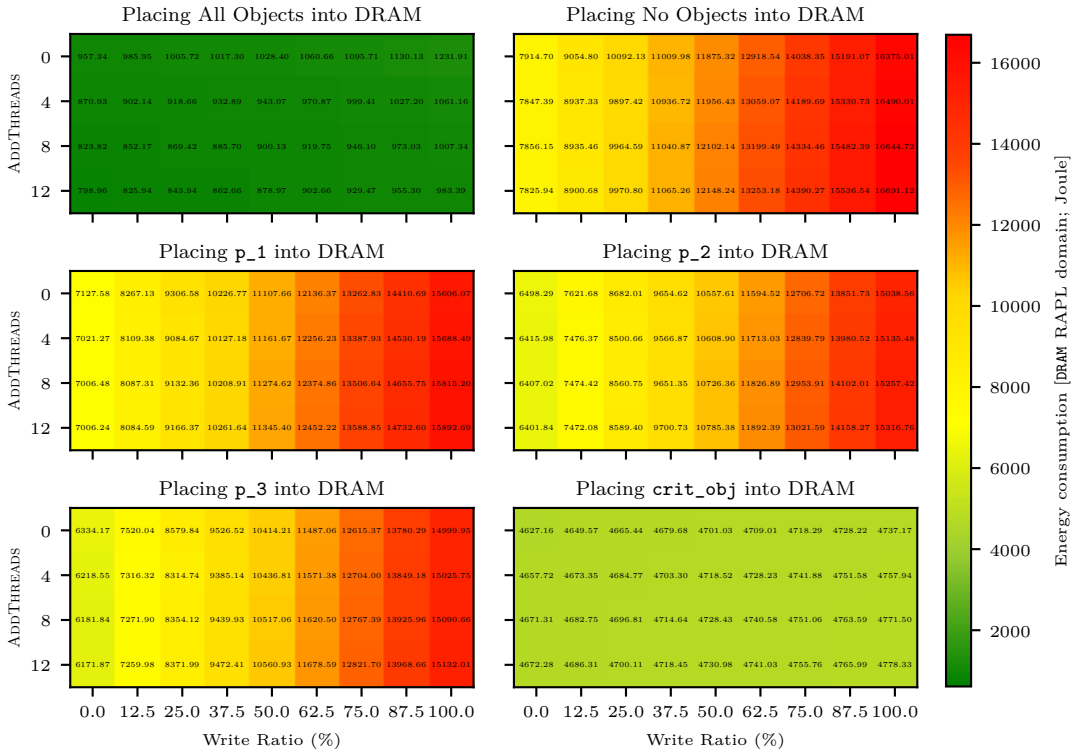
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.11.: Energy consumption (from the `DRAM` RAPL domain) when placing different objects into DRAM

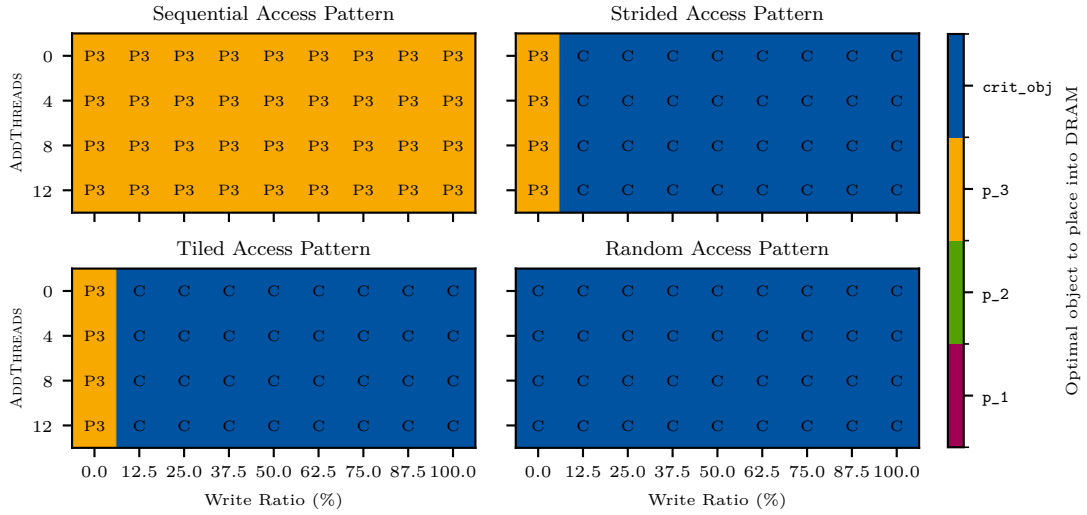
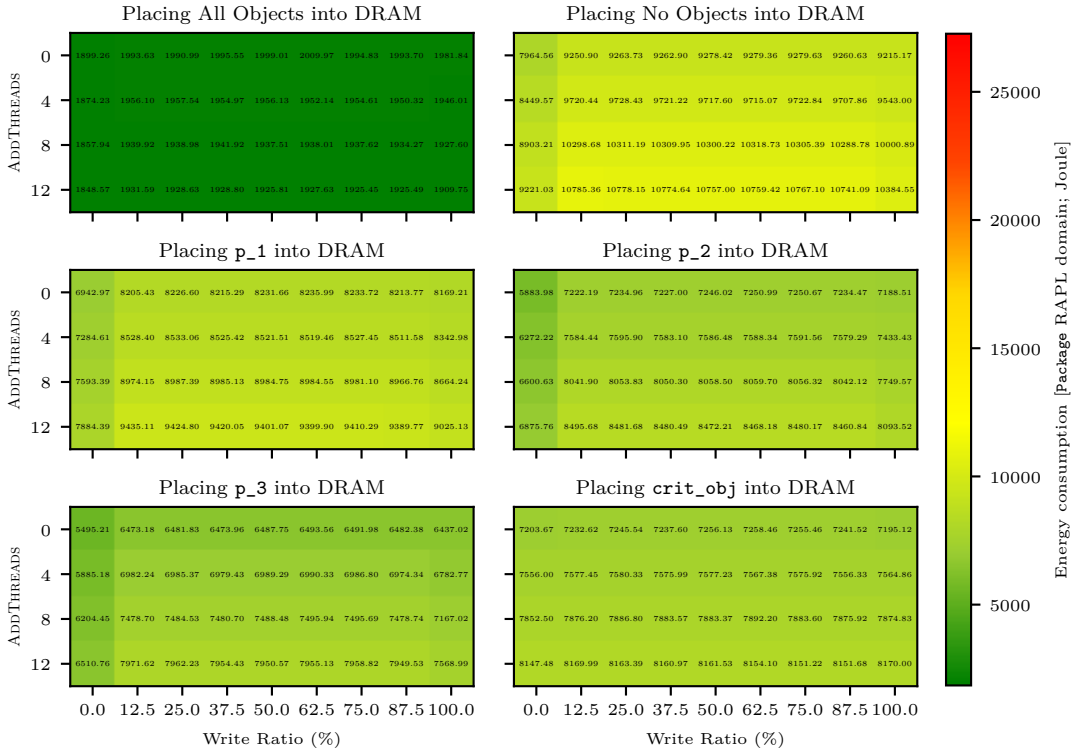
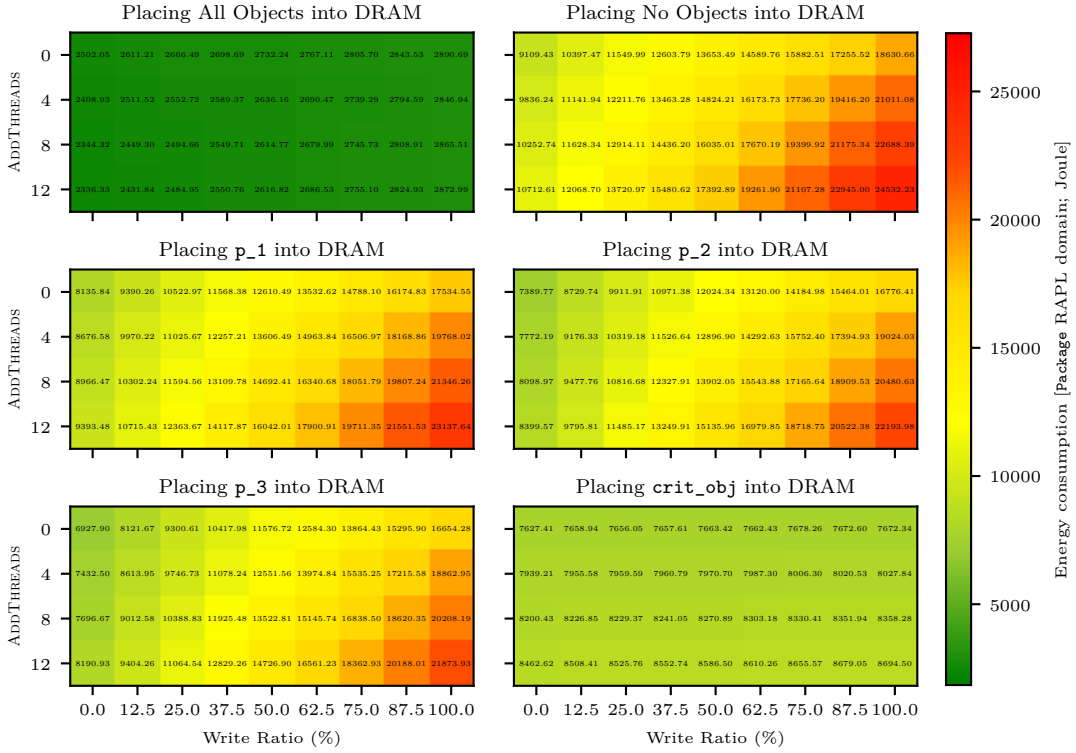


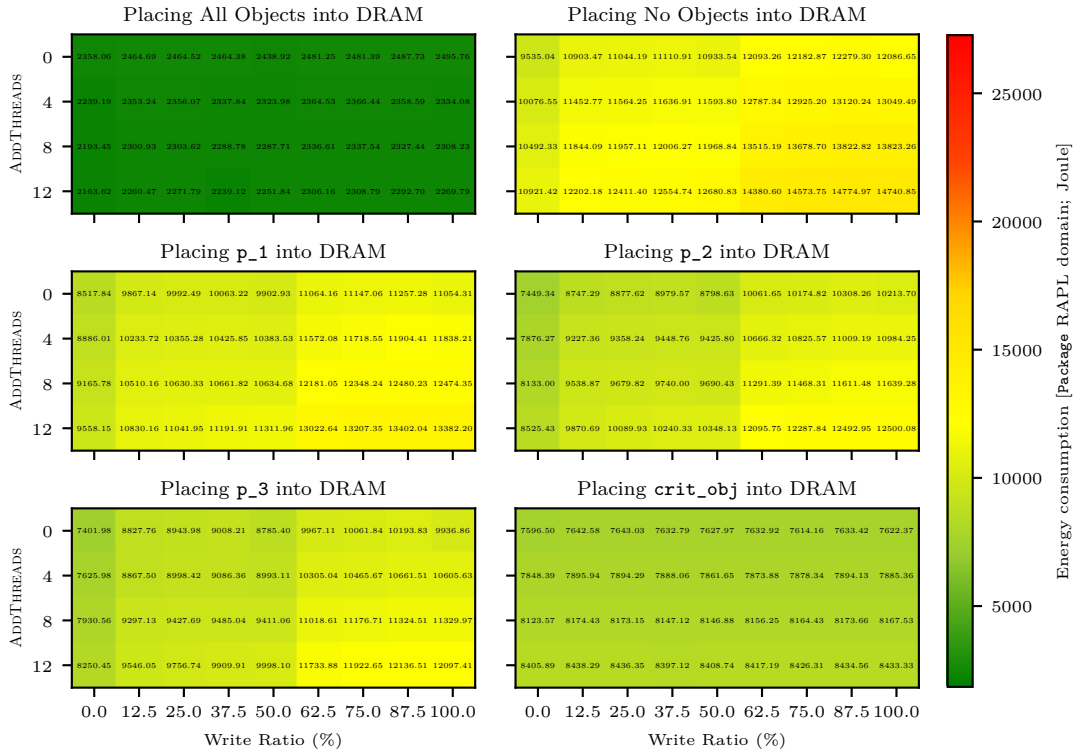
Figure A.12.: Optimal object to place into DRAM to minimize the energy consumption of the `Package` RAPL domain



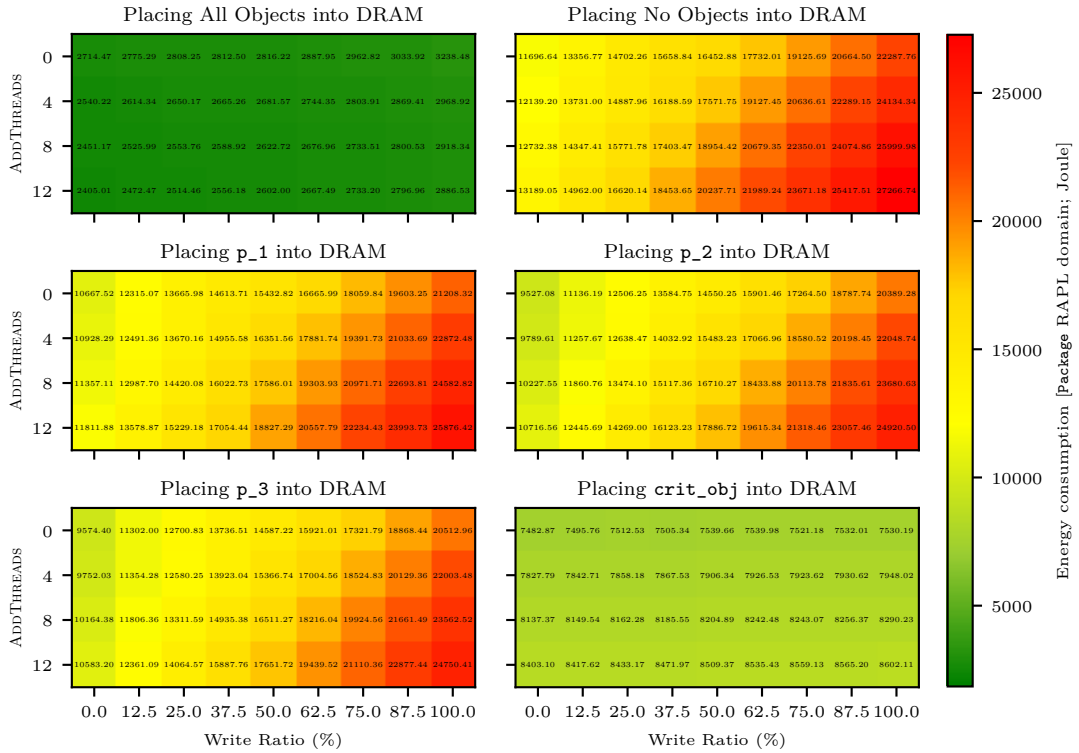
(a) Results for configurations accessing `crit_obj` with a sequential access pattern



(b) Results for configurations accessing `crit_obj` with a strided access pattern



(c) Results for configurations accessing `crit_obj` with a tiled access pattern



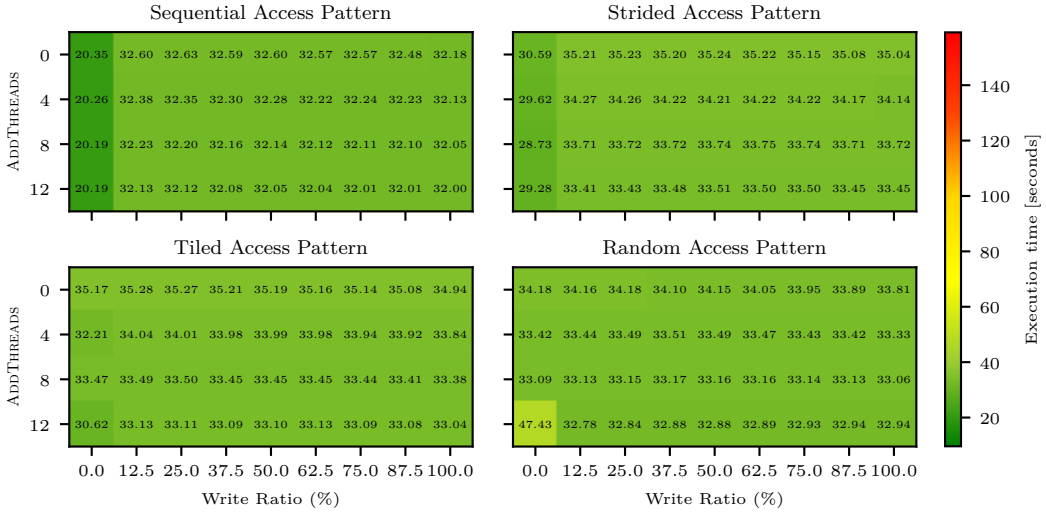
(d) Results for configurations accessing `crit_obj` with a random access pattern

Figure A.13.: Energy consumption (from the `Package` RAPL domain) when placing different objects into DRAM

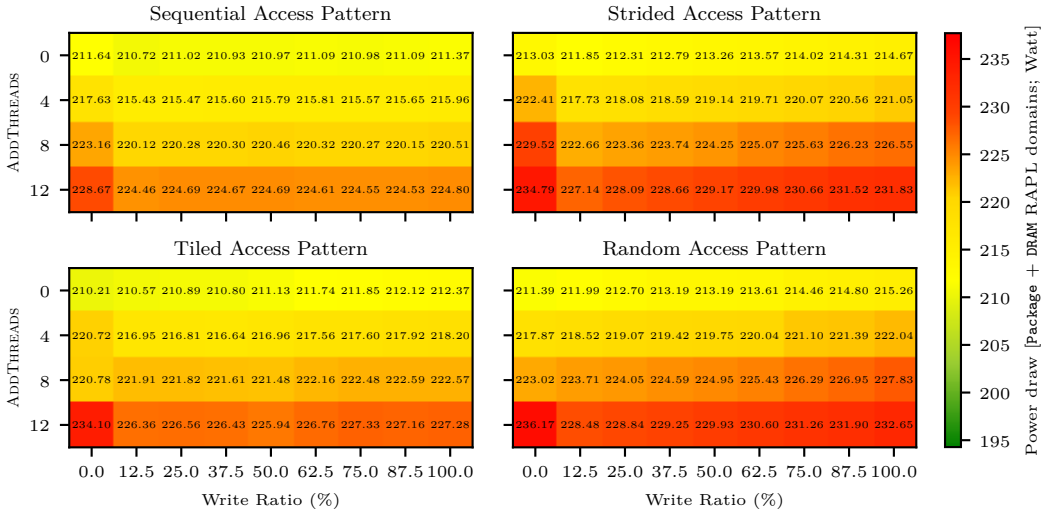
B. Framework Results for the Synthetic Benchmark

B.1. DRAM Profiling Experiment

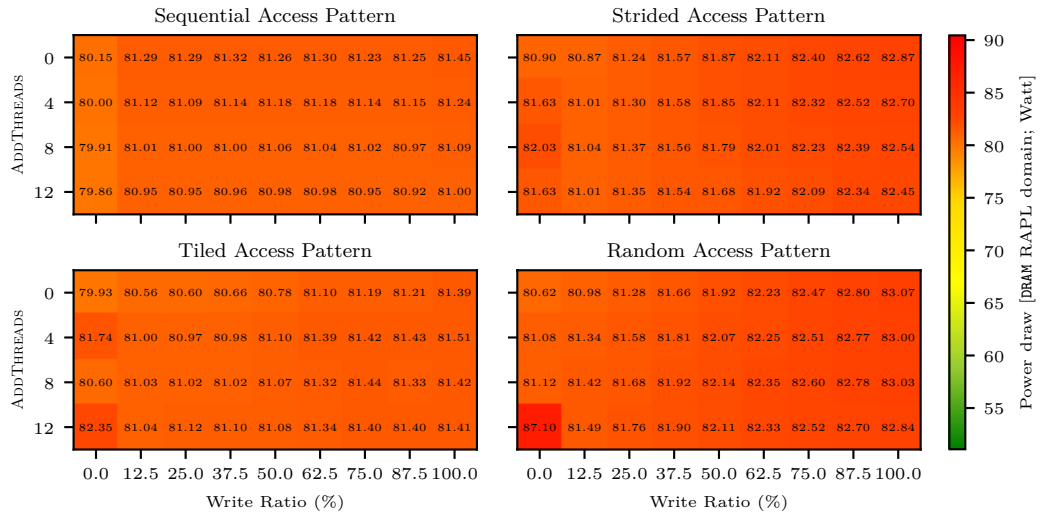
B.1.1. H2M



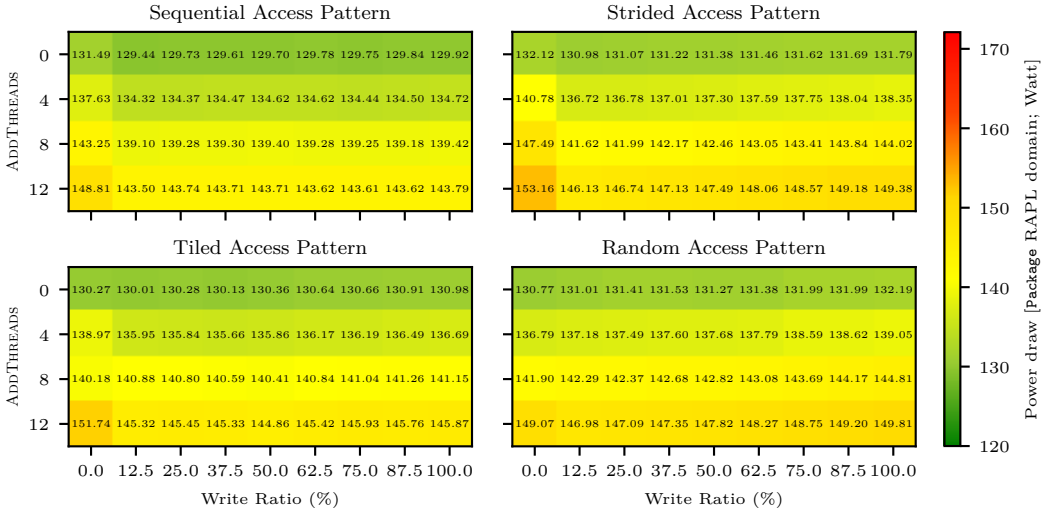
(a) Execution time



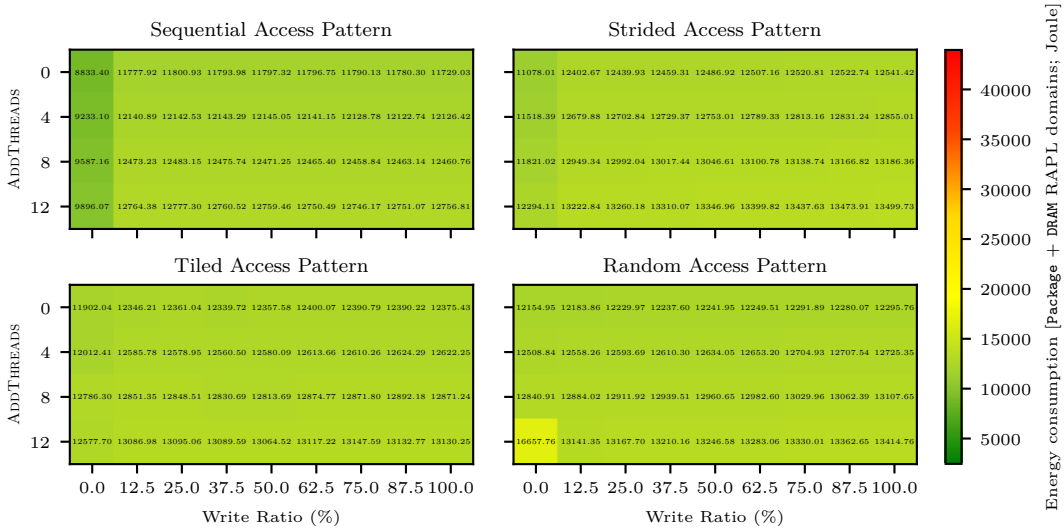
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



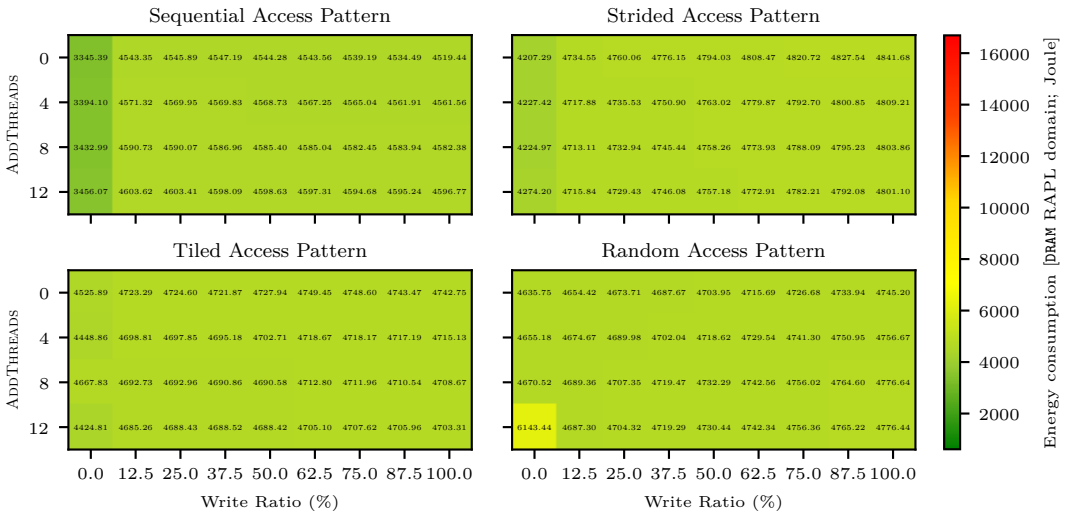
(c) Power draw (DRAM RAPL domain)



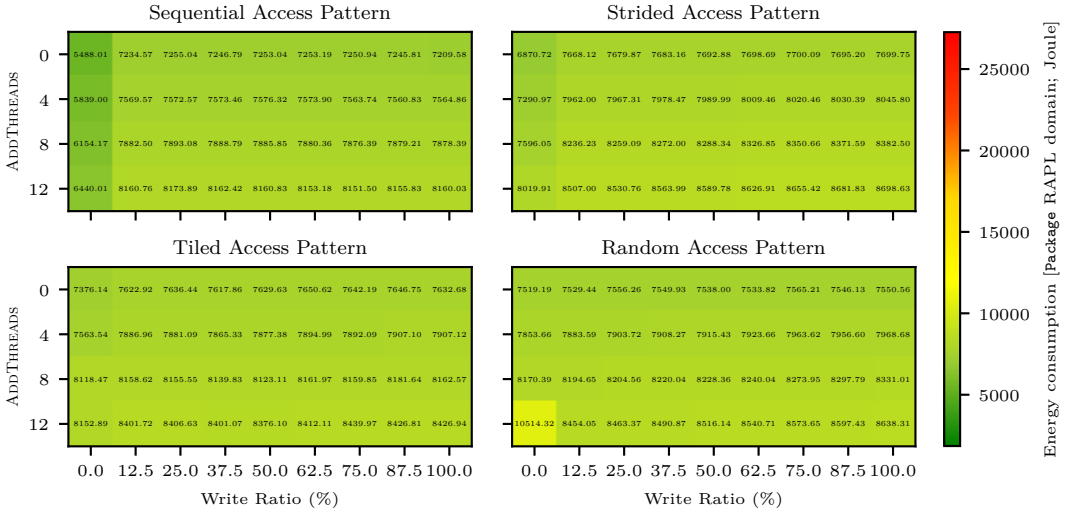
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

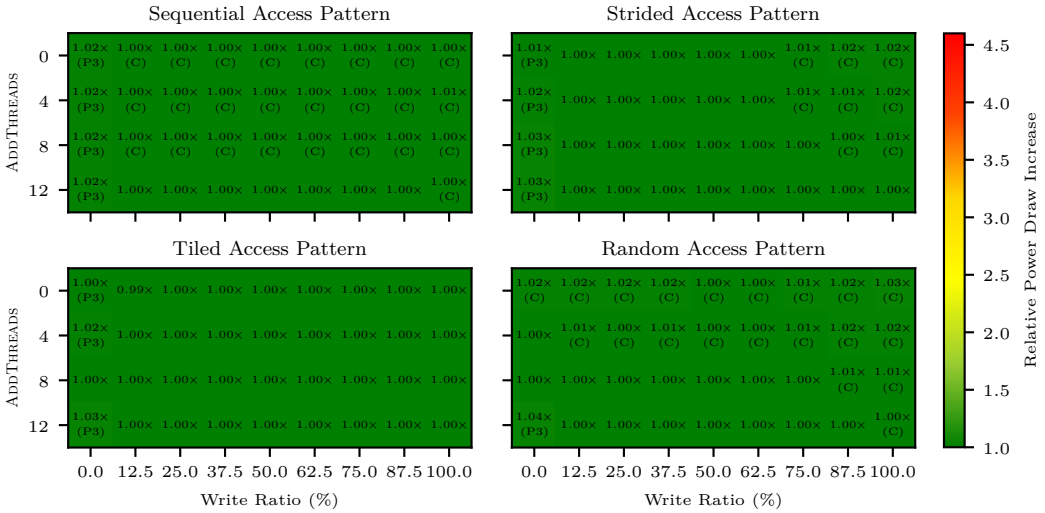


(f) Energy consumption (DRAM RAPL domain)

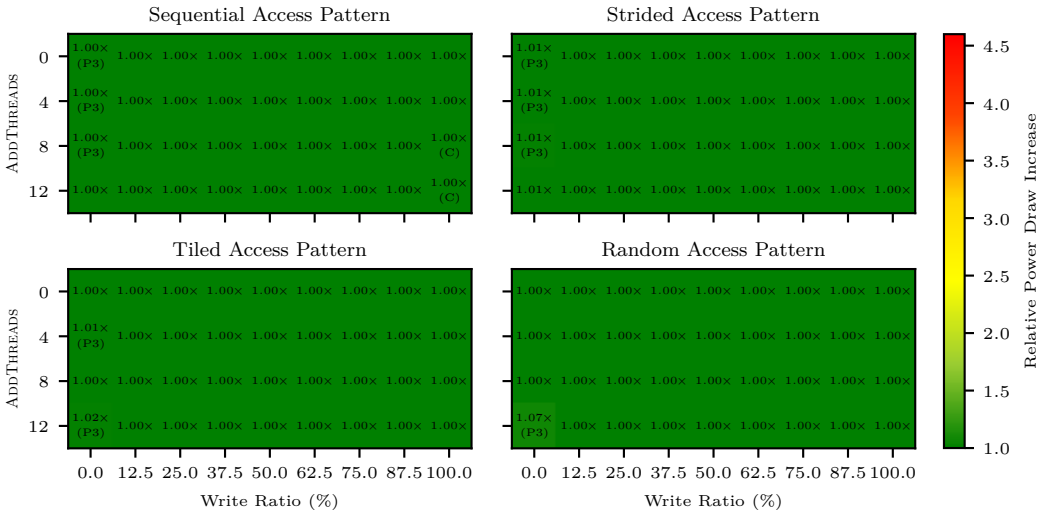


(g) Energy consumption (Package RAPL domain)

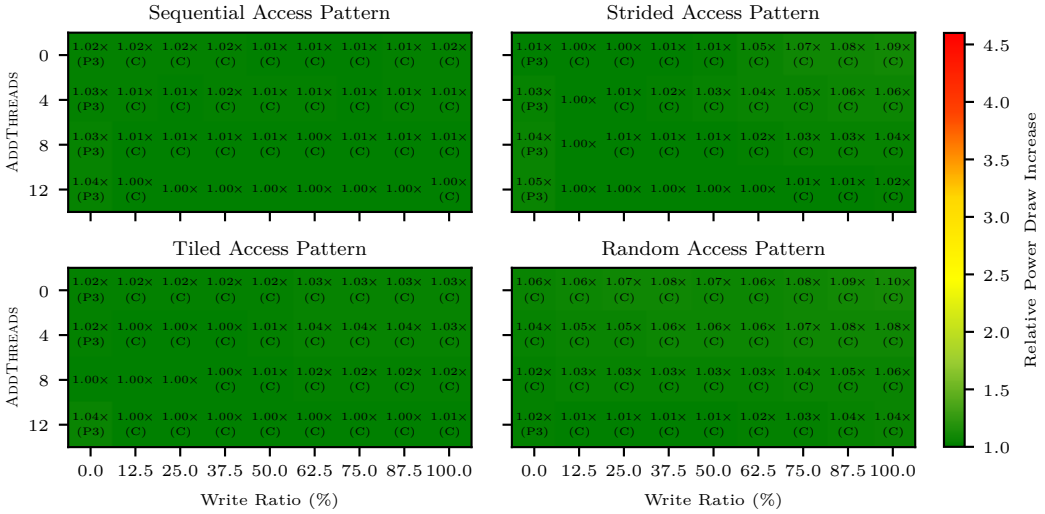
Figure B.1.: Execution time, power draw, and energy consumption of H2M's placement decisions



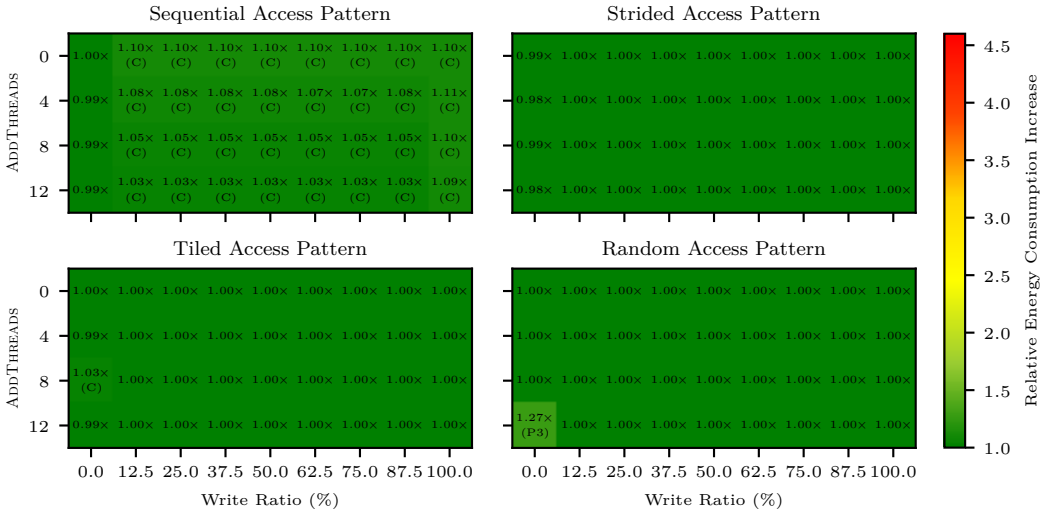
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



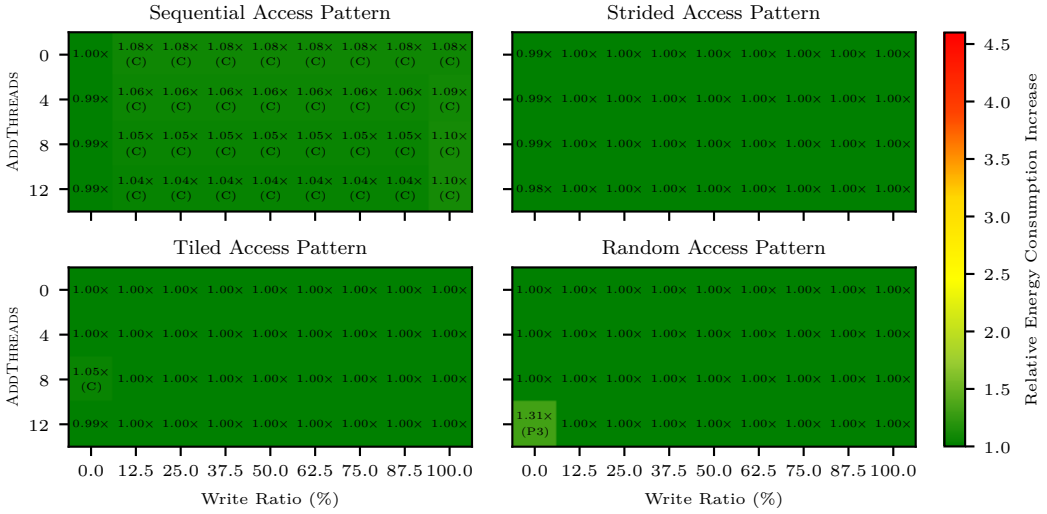
(b) Power draw overhead for the `DRAM` RAPL domain



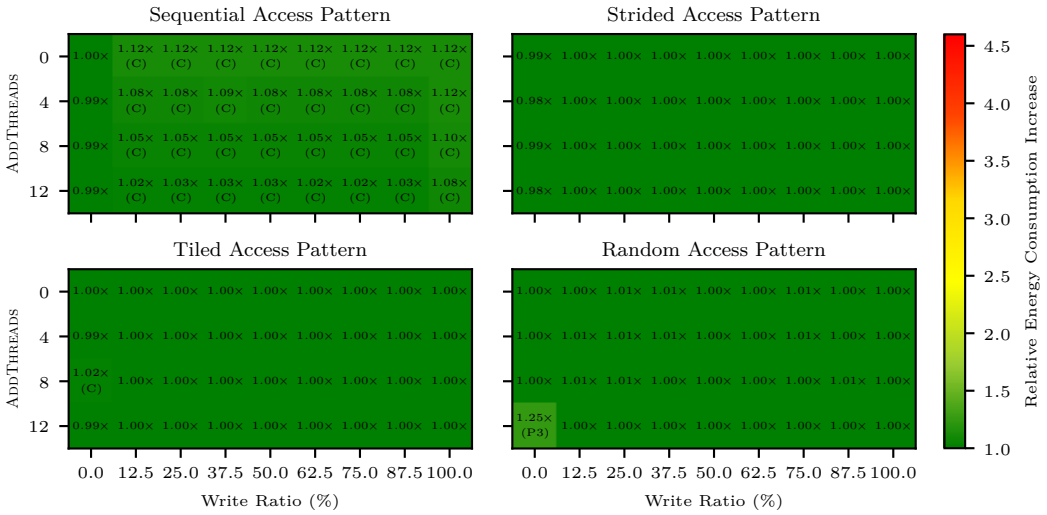
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



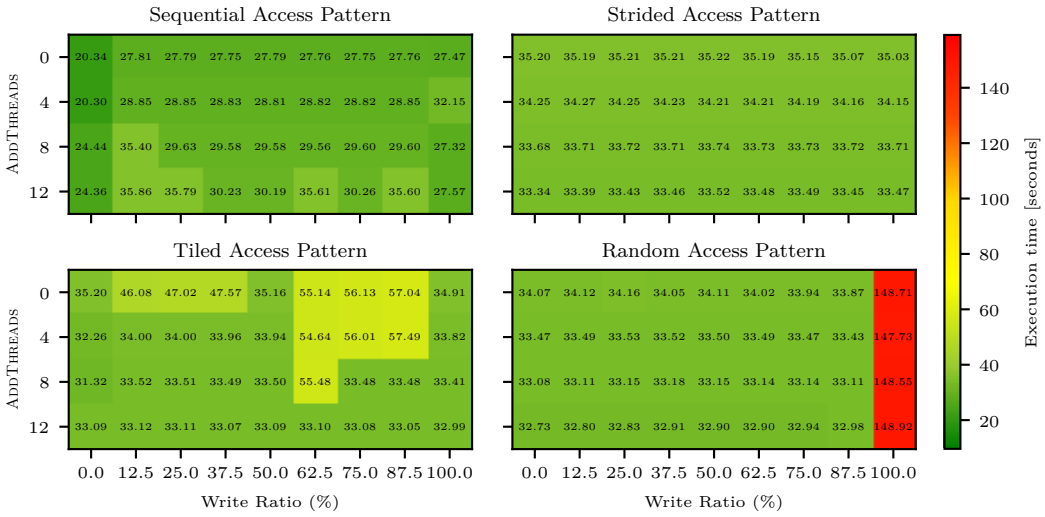
(e) Energy consumption overhead for the `DRAM` RAPL domain



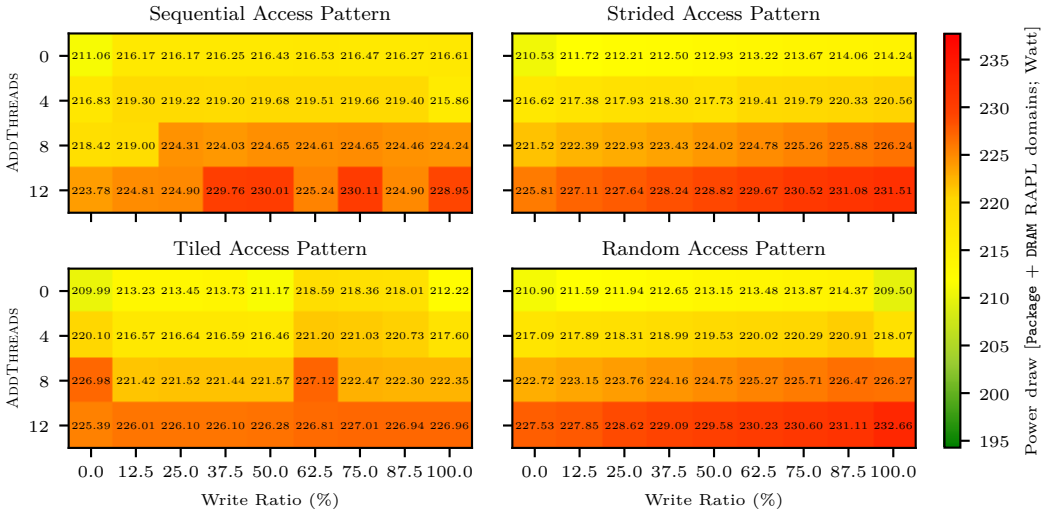
(f) Energy consumption overhead for the `Package` RAPL domain

Figure B.2.: Power draw and energy consumption overheads of H2M's placement decisions over the optimal placements

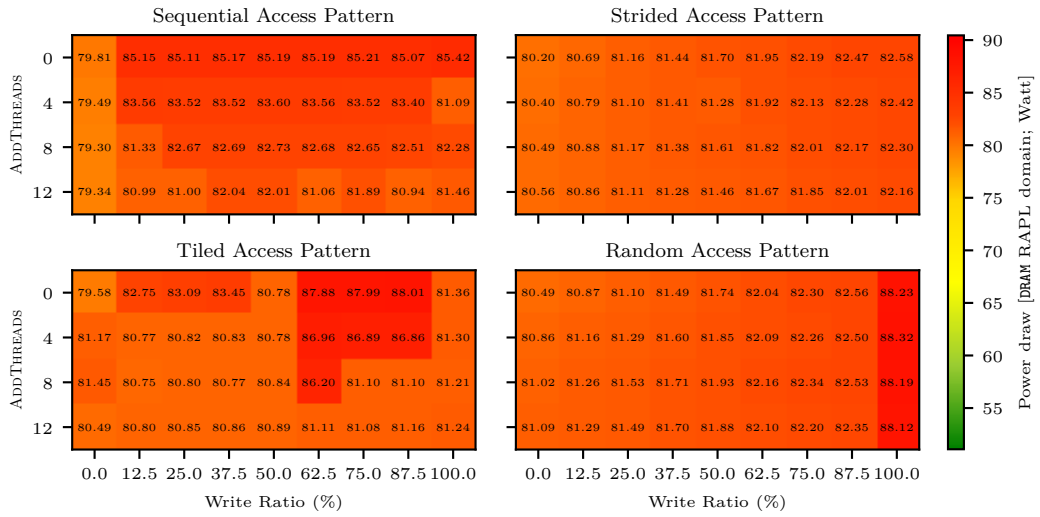
B.1.2. ecoHMEM



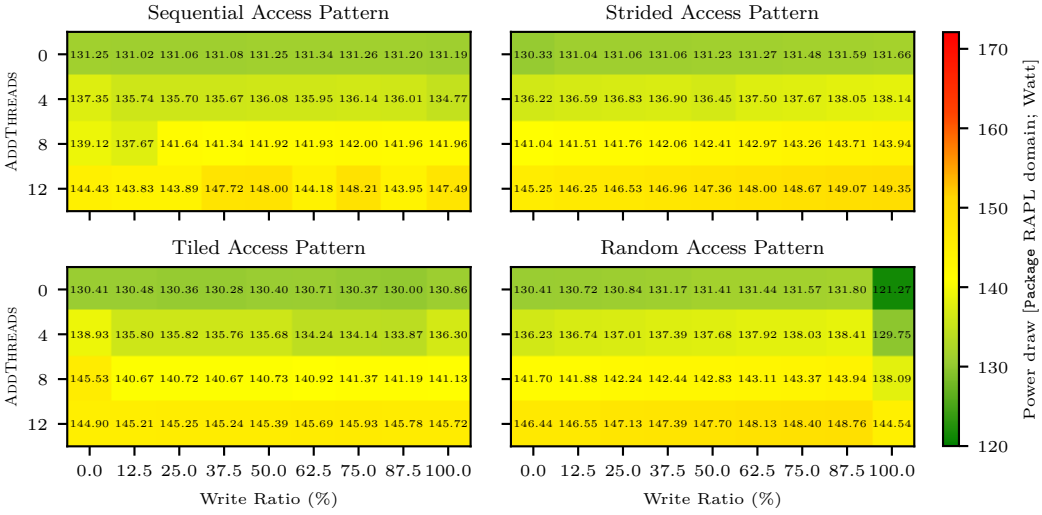
(a) Execution time



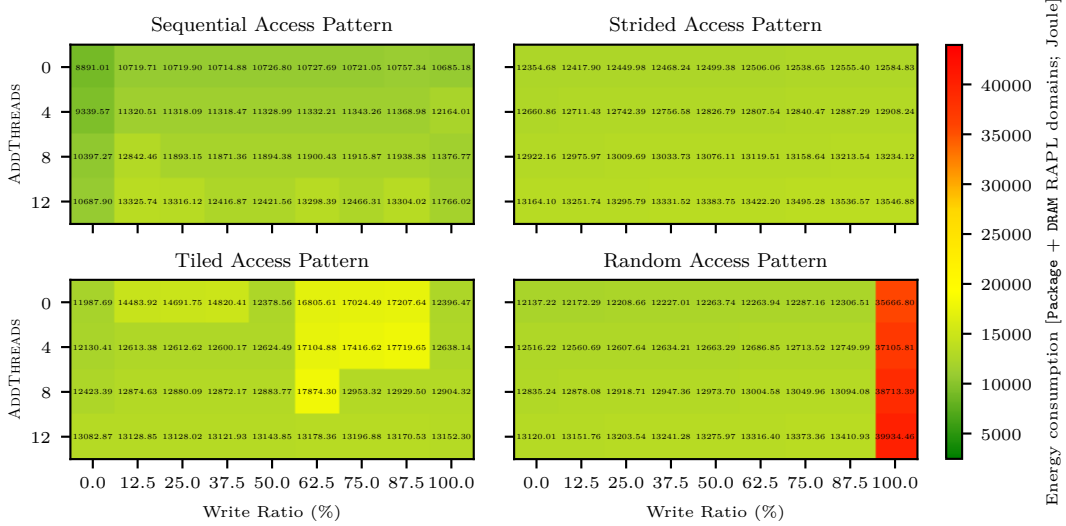
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



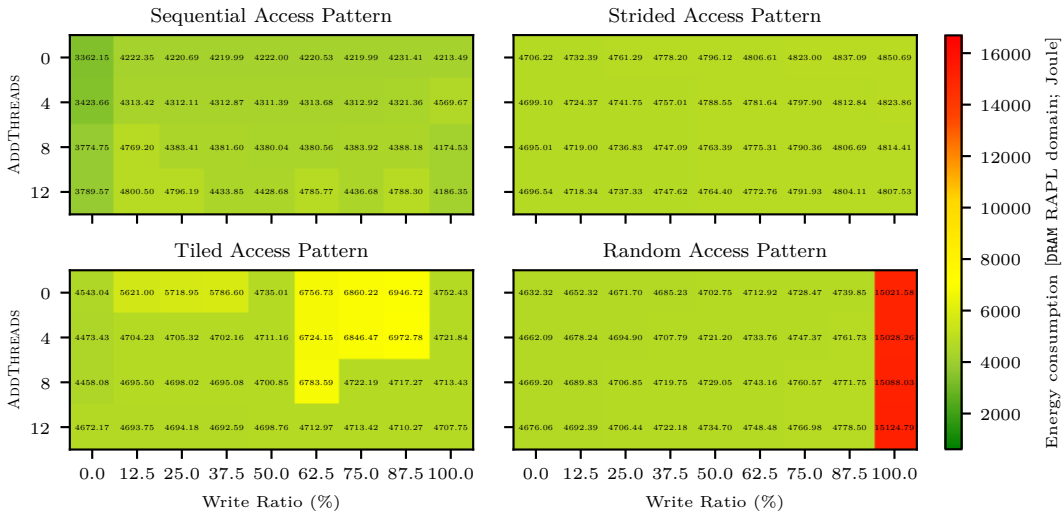
(c) Power draw (DRAM RAPL domain)



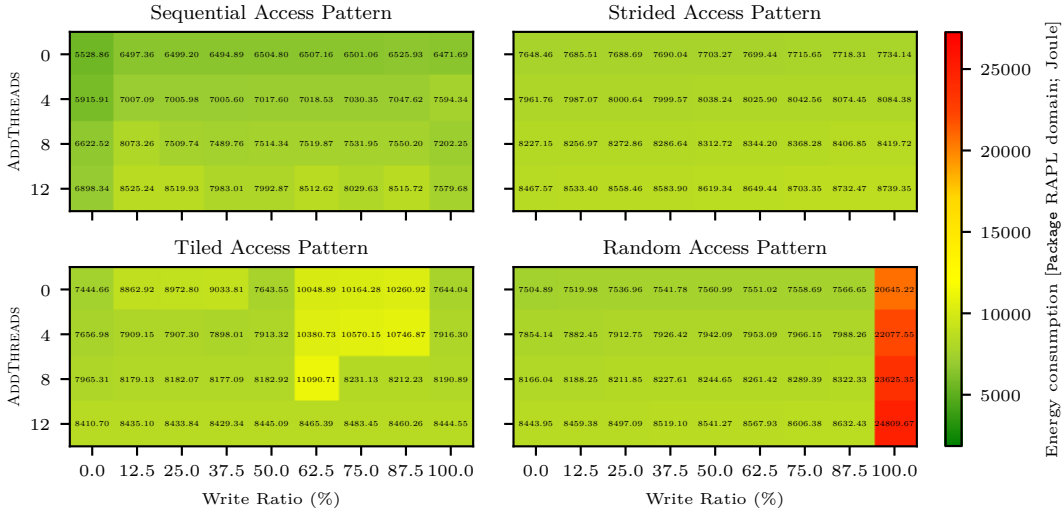
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

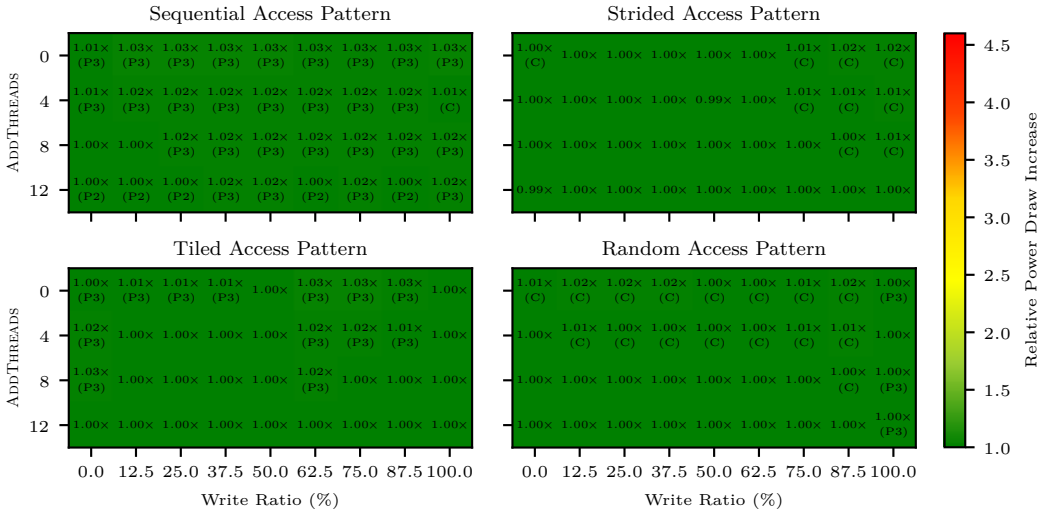


(f) Energy consumption (`DRAM` RAPL domain)

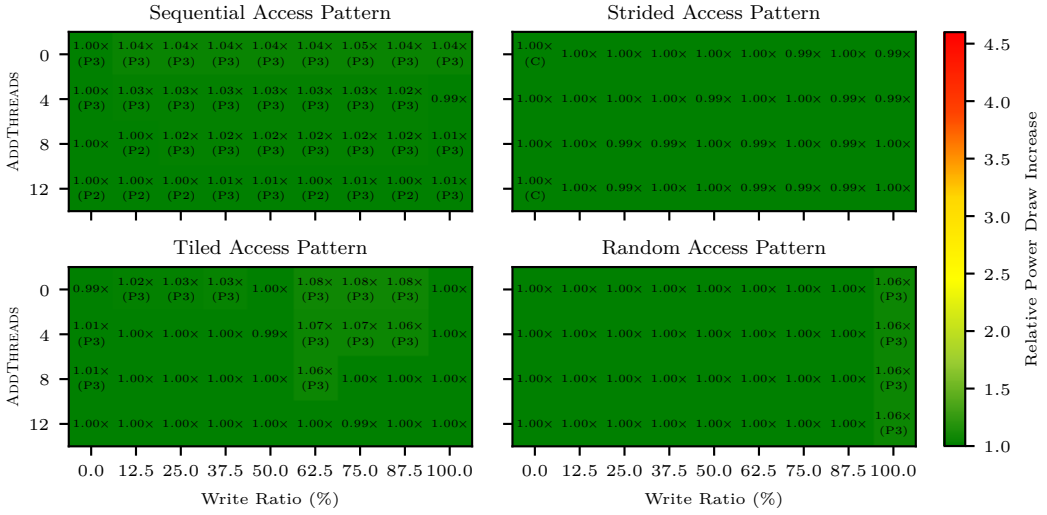


(g) Energy consumption (`Package` RAPL domain)

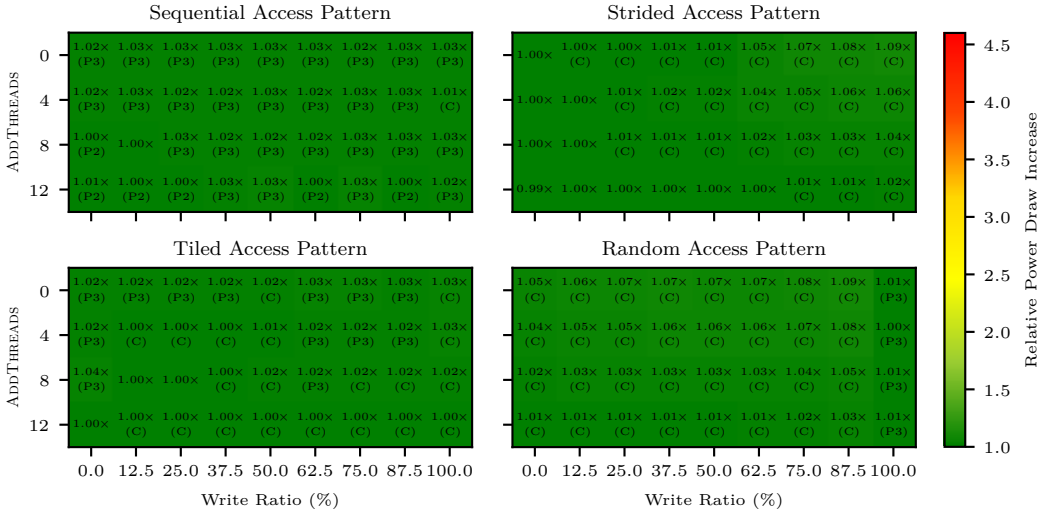
Figure B.3.: Execution time, power draw, and energy consumption of ecoHMEM's placement decisions



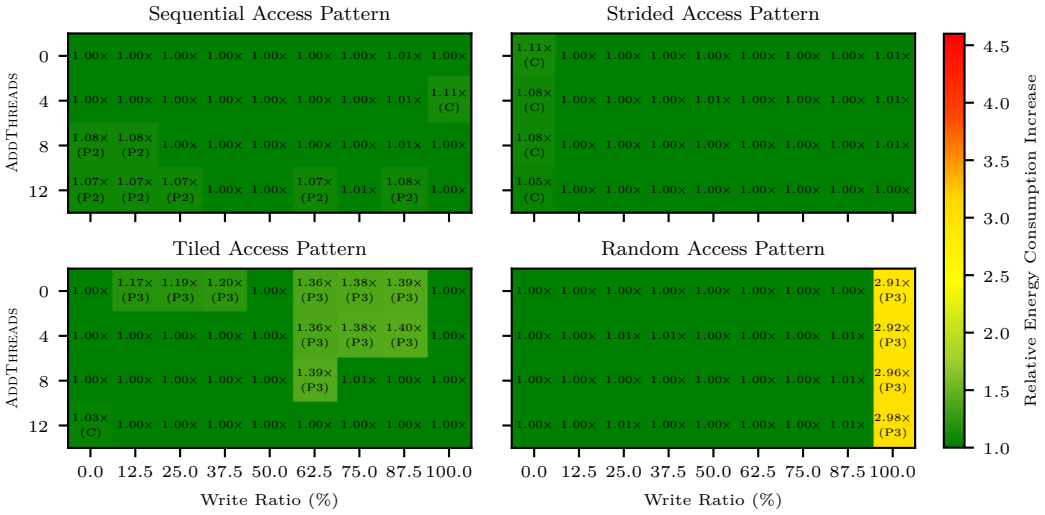
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



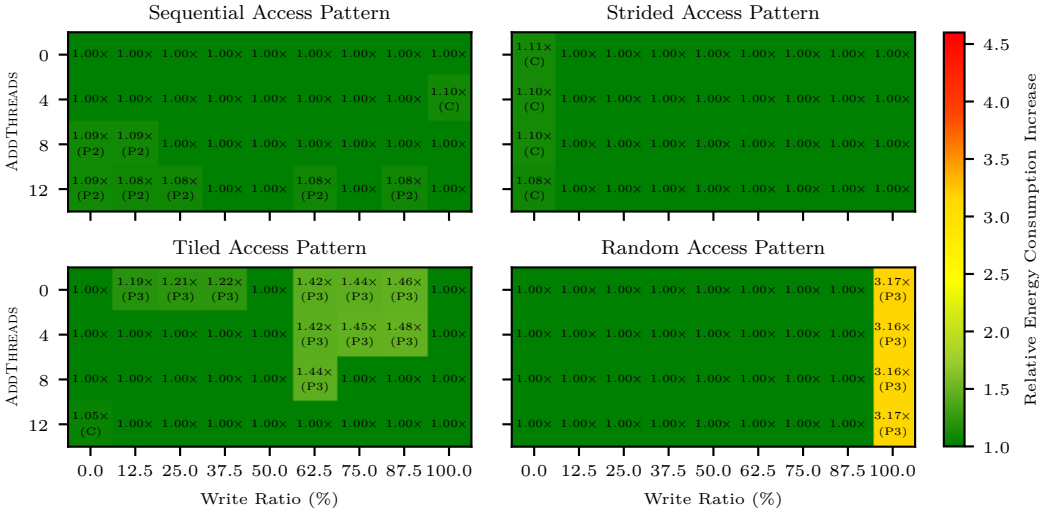
(b) Power draw overhead for the `DRAM` RAPL domain



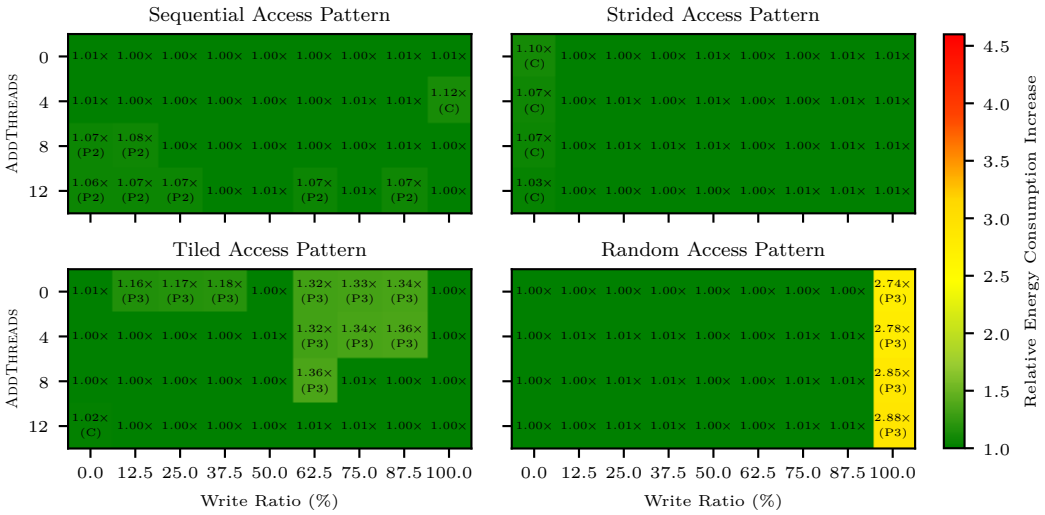
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



(e) Energy consumption overhead for the `DRAM` RAPL domain

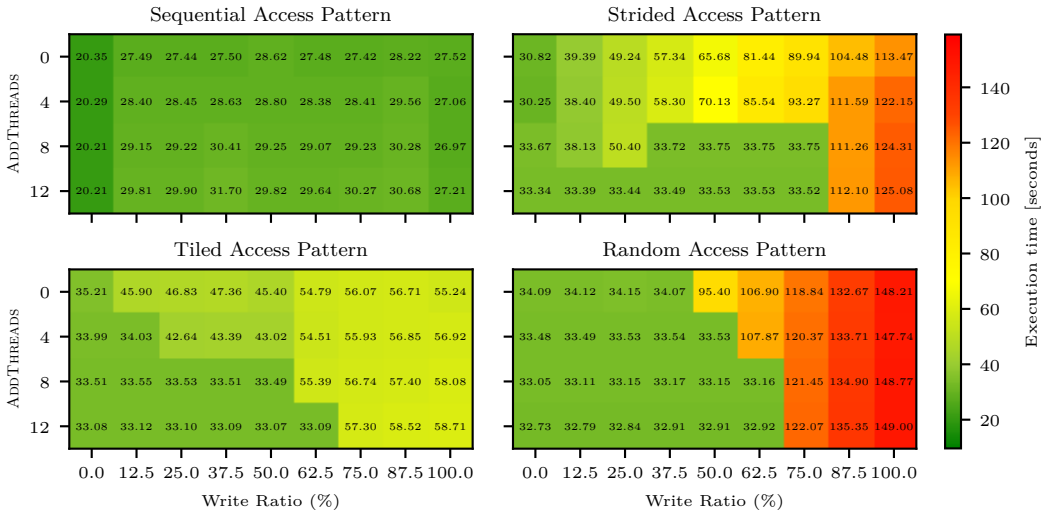


(f) Energy consumption overhead for the `Package` RAPL domain

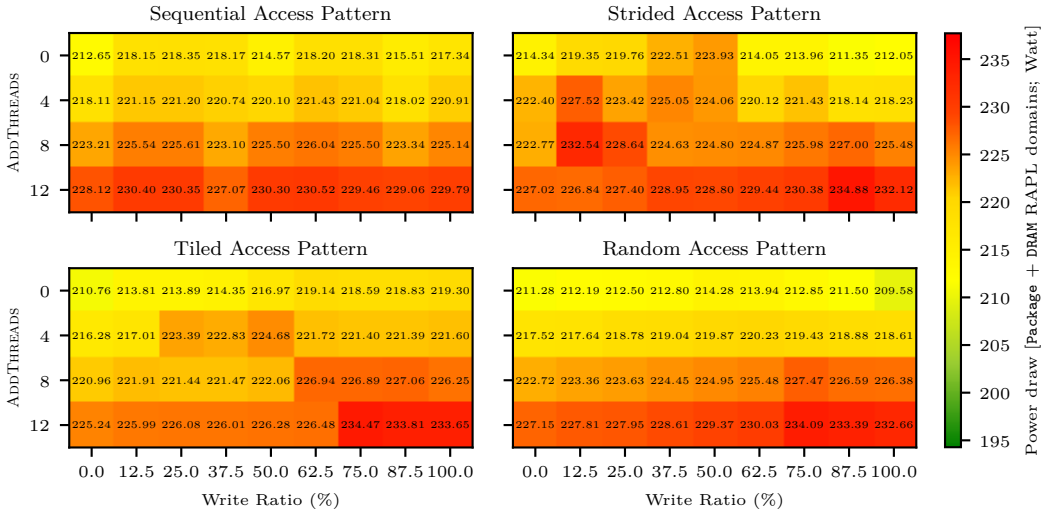
Figure B.4.: Power draw and energy consumption overheads of ecoHMEM's placement decisions over the optimal placements

B.1.3. ProfDP

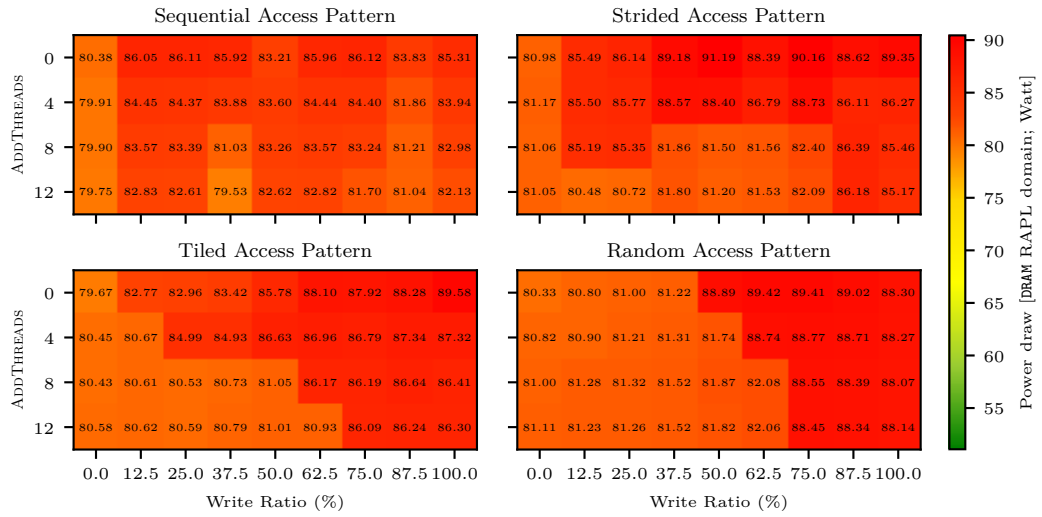
Latency-Aware Placement Optimization



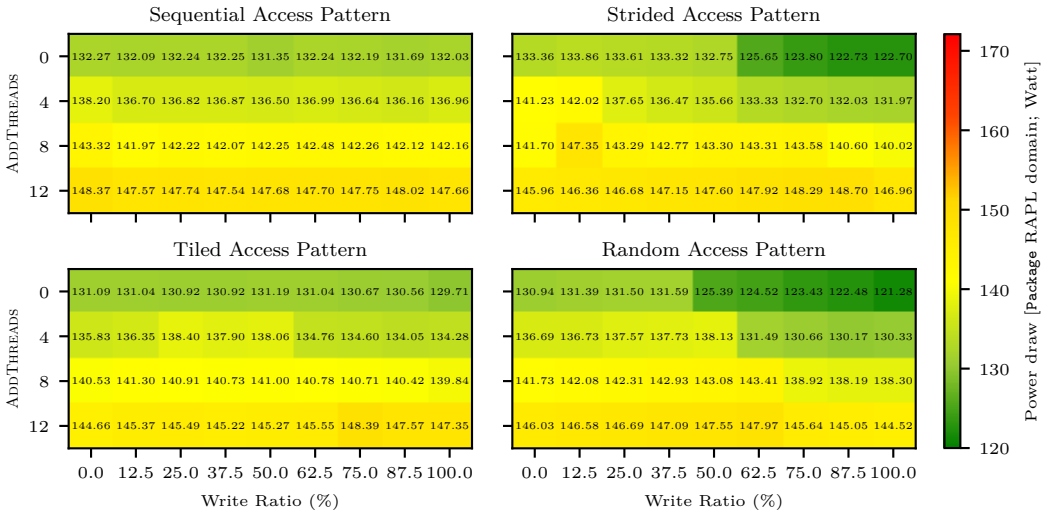
(a) Execution time



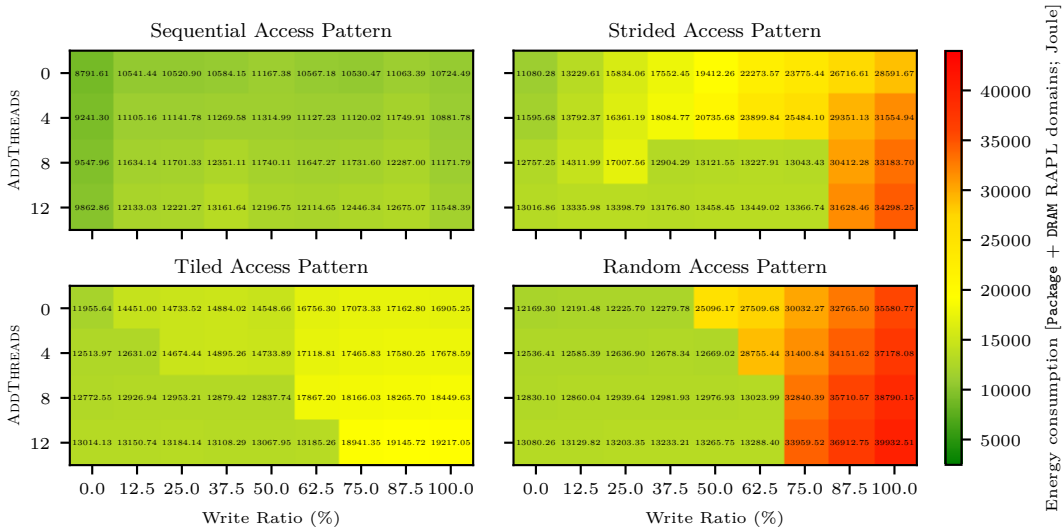
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



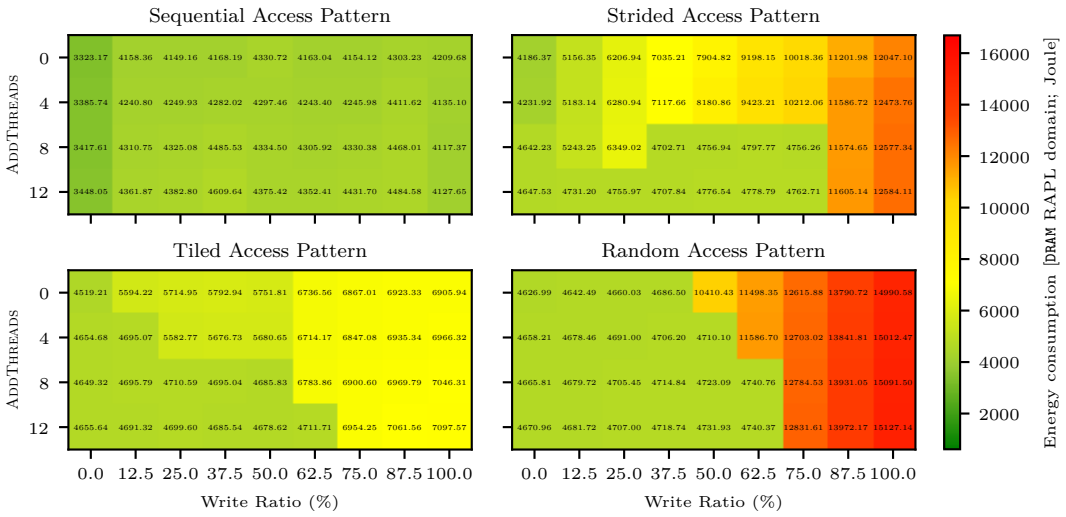
(c) Power draw (DRAM RAPL domain)



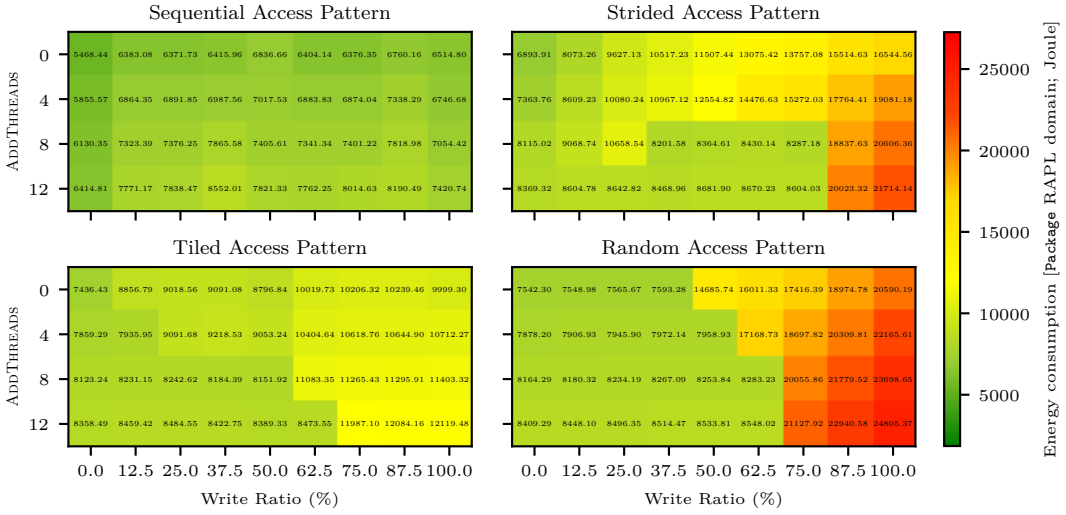
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

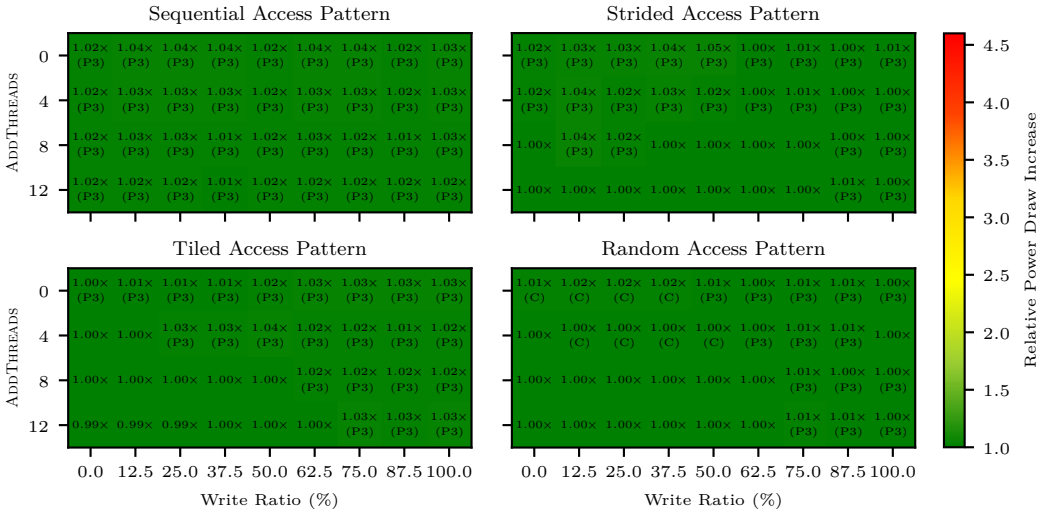


(f) Energy consumption (DRAM RAPL domain)

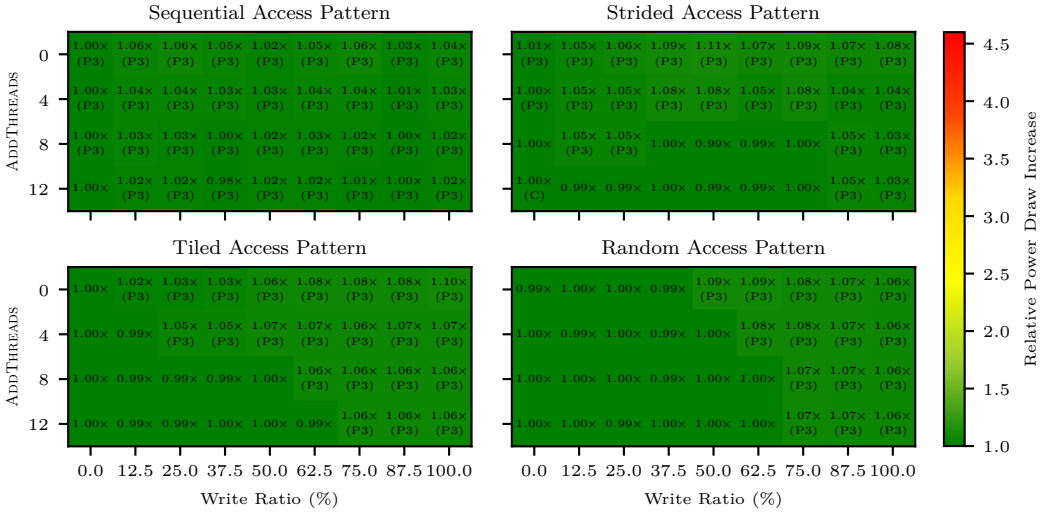


(g) Energy consumption (Package RAPL domain)

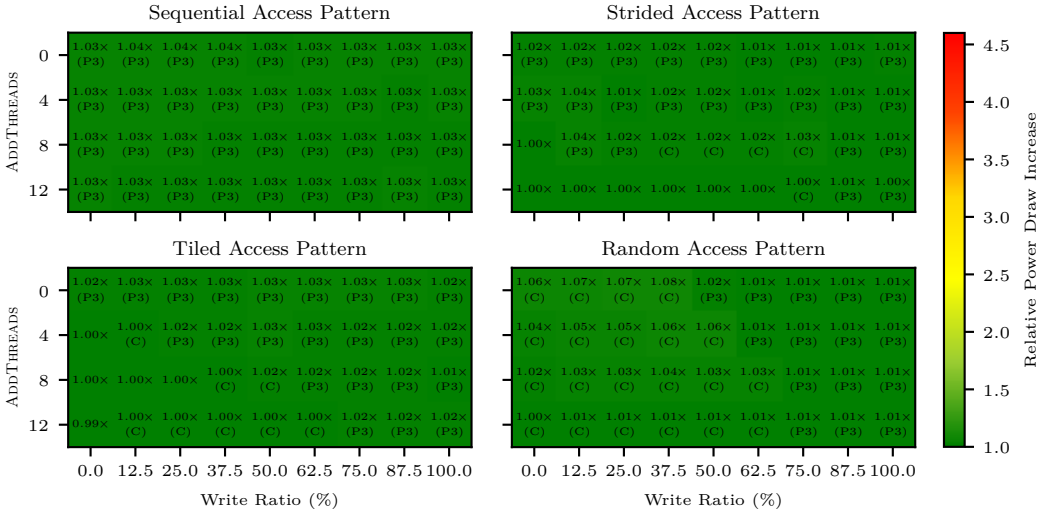
Figure B.5.: Execution time, power draw, and energy consumption of ProfDP's latency-aware placement decisions



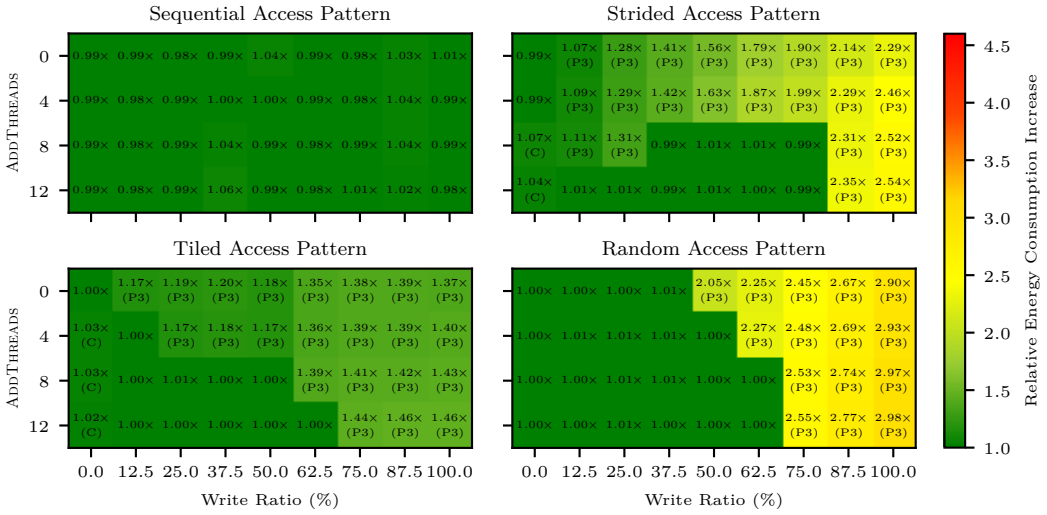
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



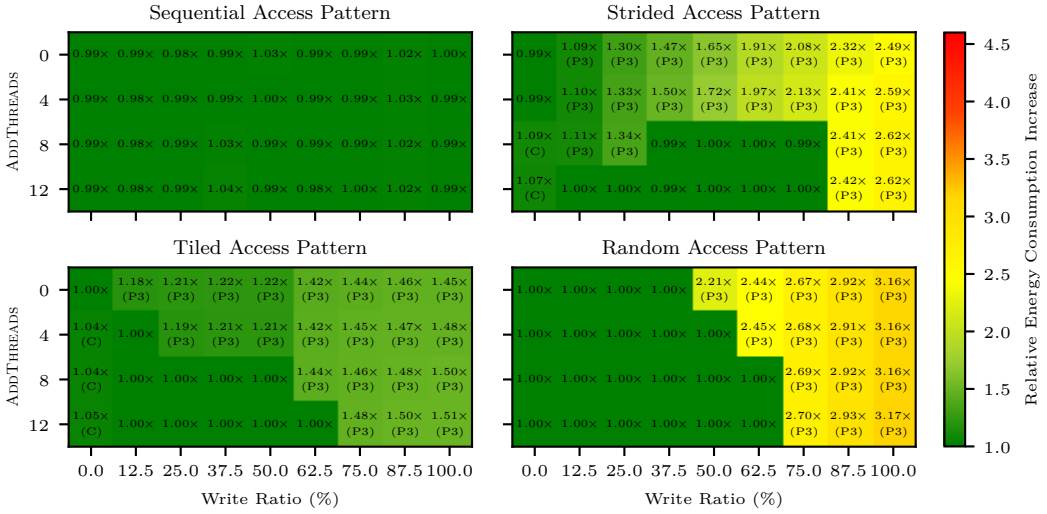
(b) Power draw overhead for the `DRAM` RAPL domain



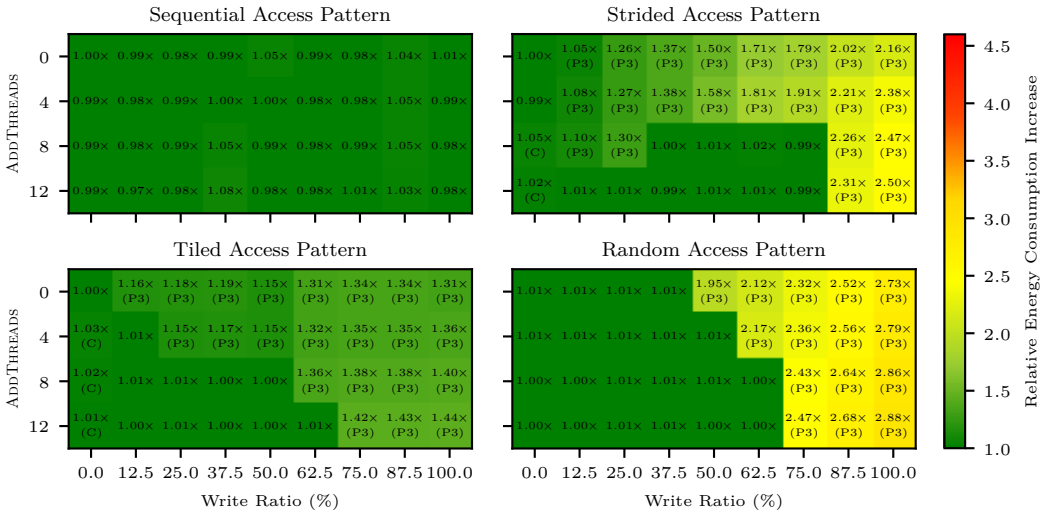
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



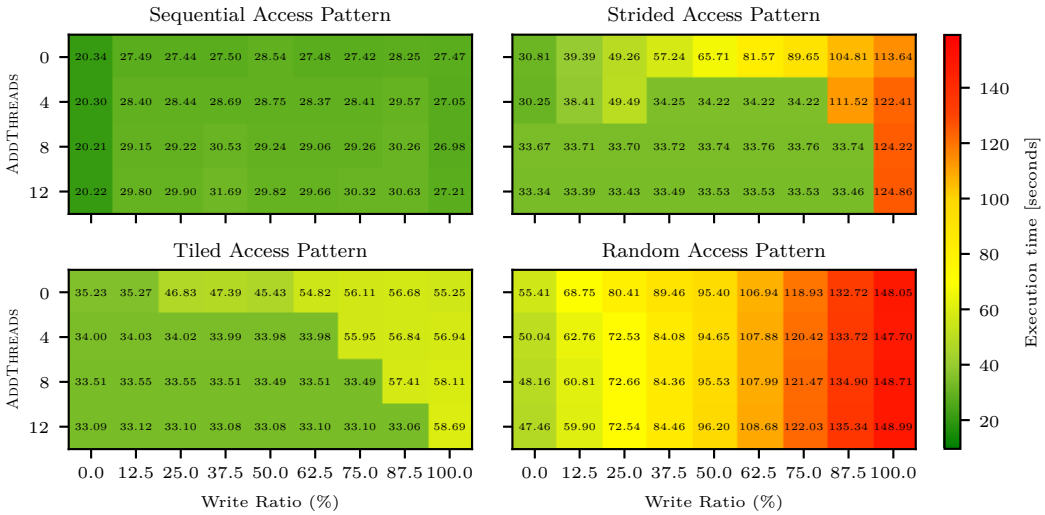
(e) Energy consumption overhead for the `DRAM` RAPL domain



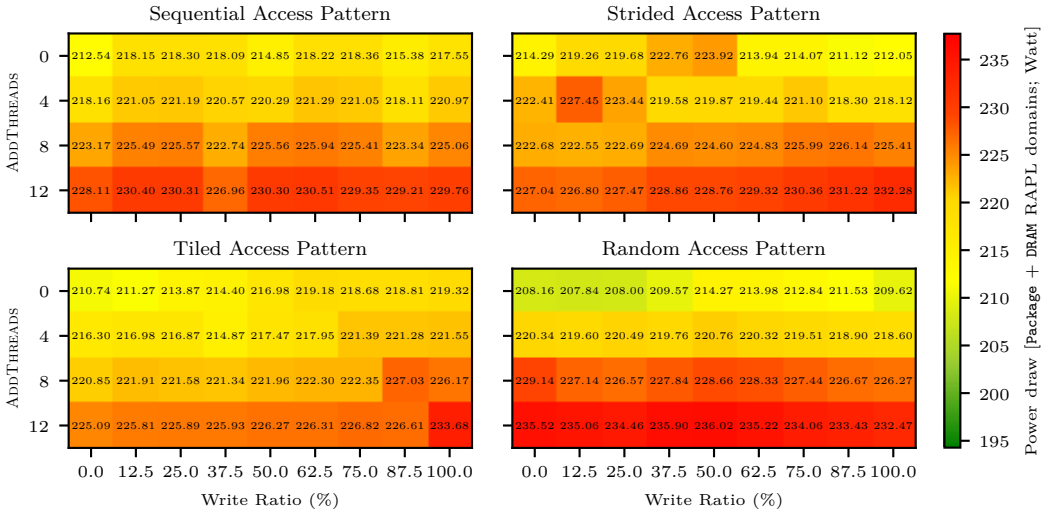
(f) Energy consumption overhead for the `Package` RAPL domain

Figure B.6.: Power draw and energy consumption overheads of ProfDP's latency-aware placement decisions over the optimal placements

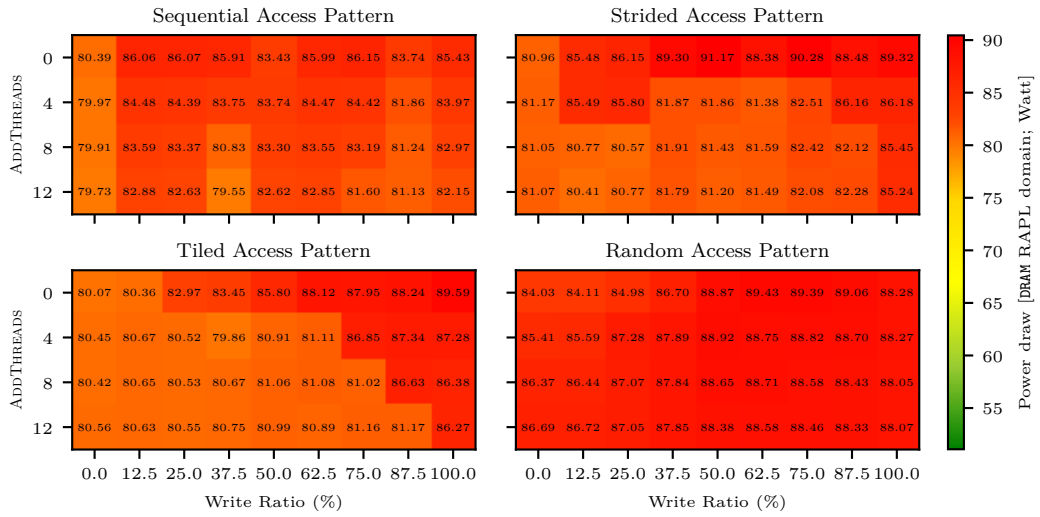
Bandwidth-Aware Placement Optimization



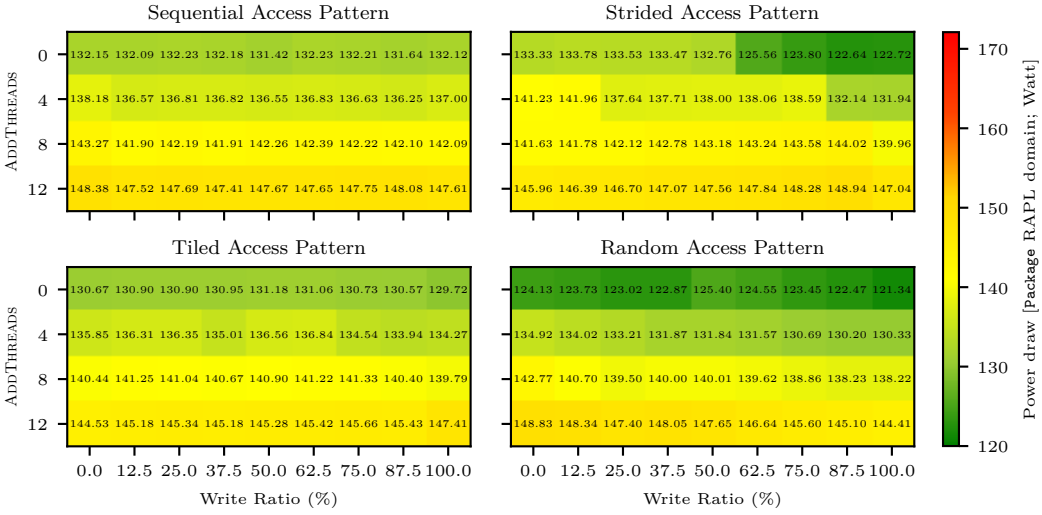
(a) Execution time



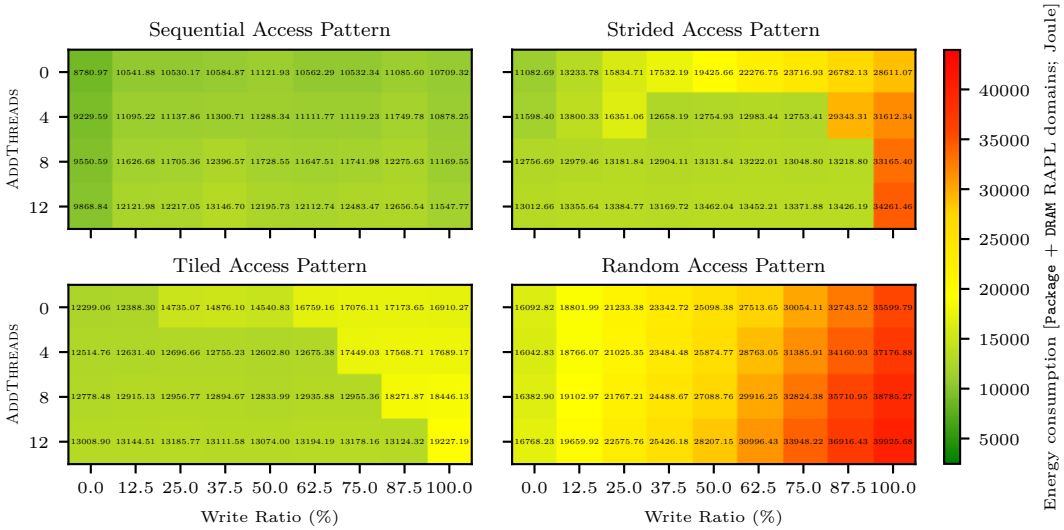
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



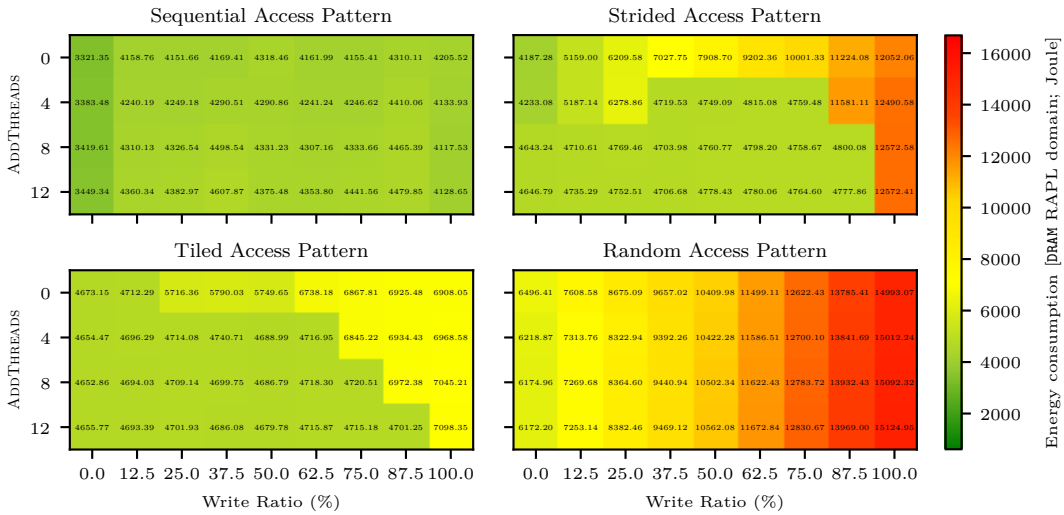
(c) Power draw (DRAM RAPL domain)



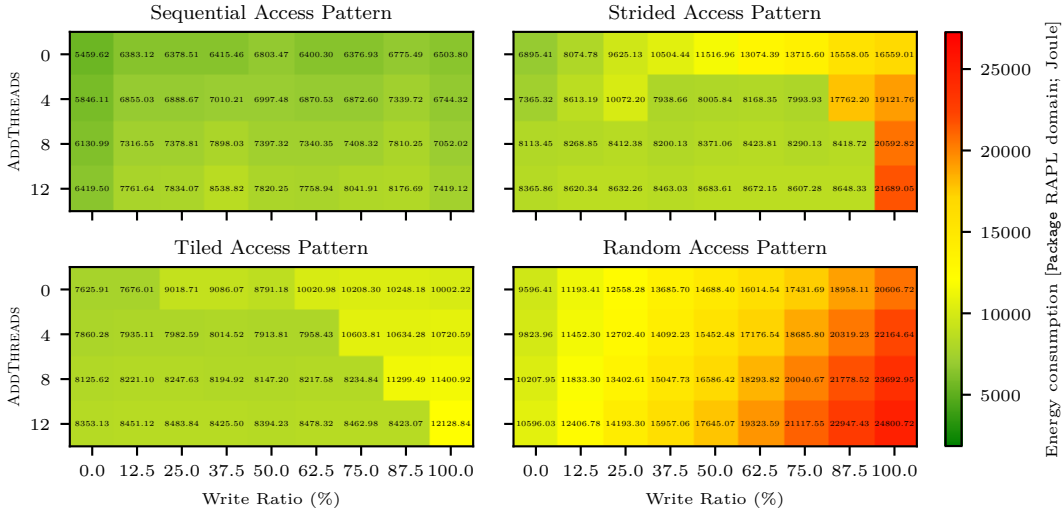
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

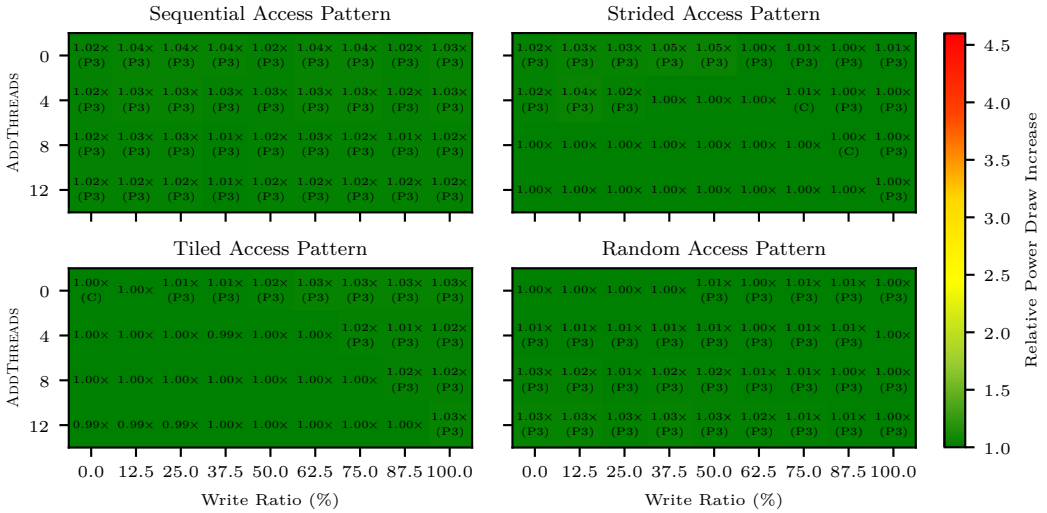


(f) Energy consumption (DRAM RAPL domain)

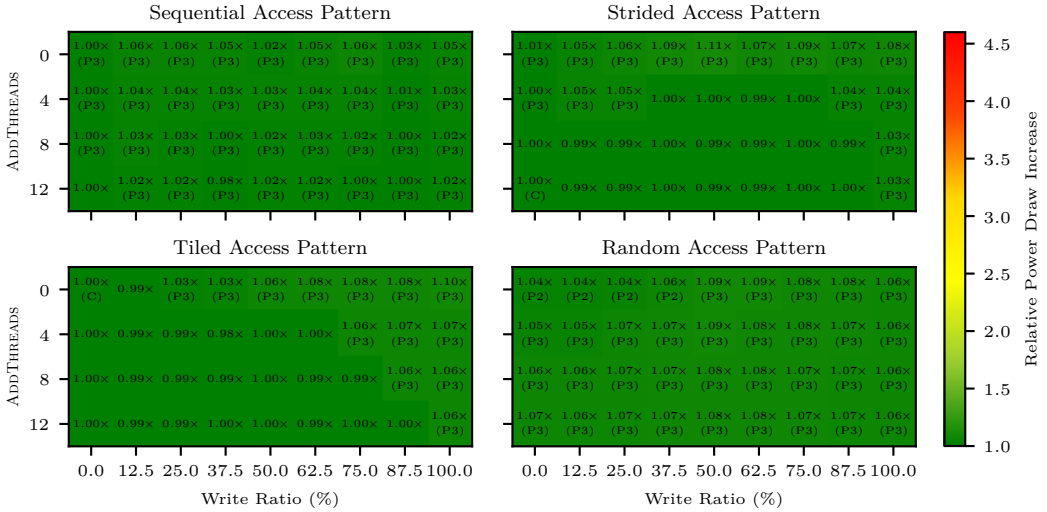


(g) Energy consumption (Package RAPL domain)

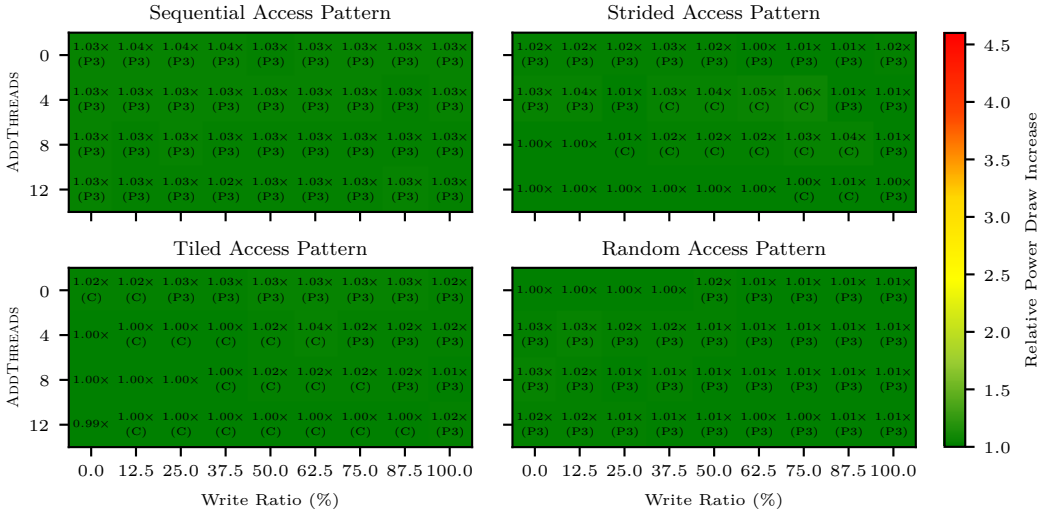
Figure B.7.: Execution time, power draw, and energy consumption of ProfDP's bandwidth-aware placement decisions



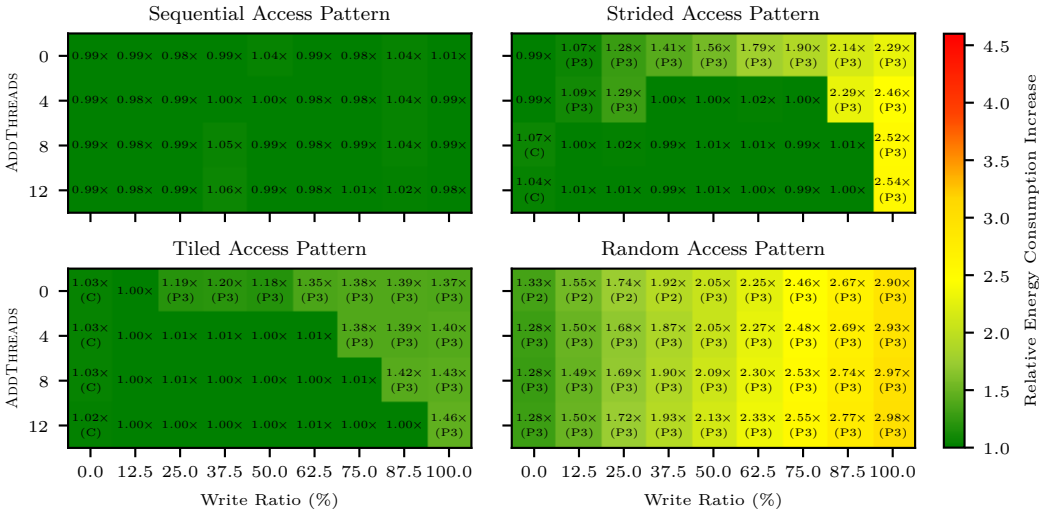
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



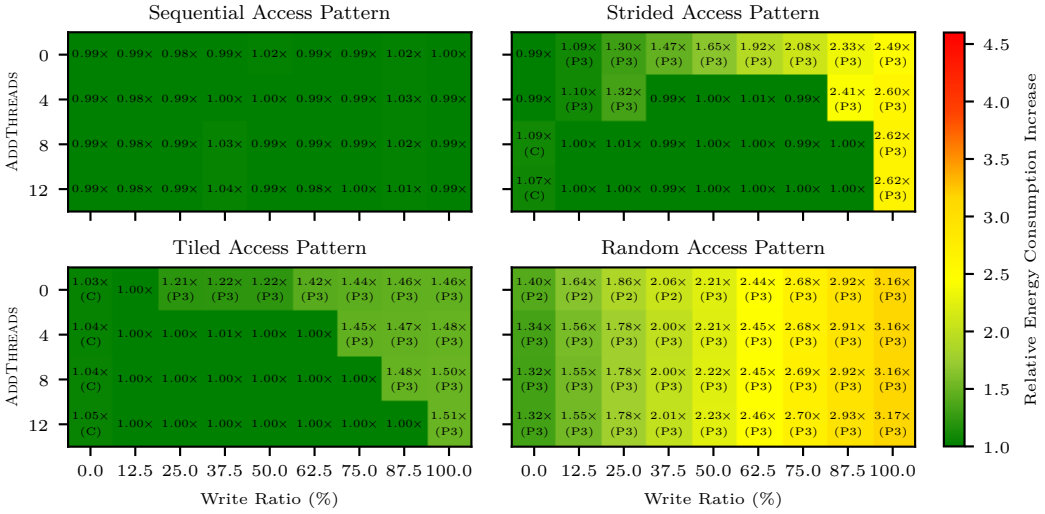
(b) Power draw overhead for the `DRAM` RAPL domain



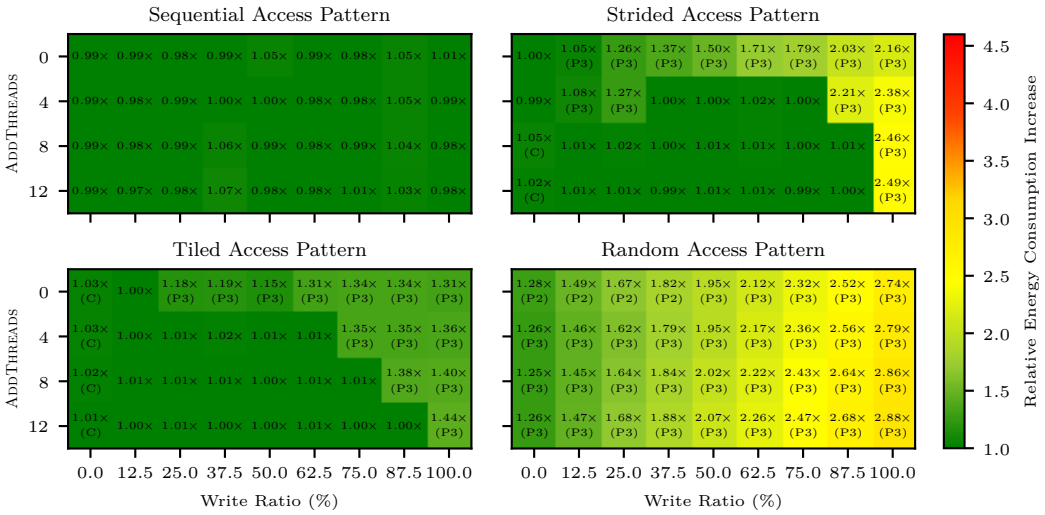
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



(e) Energy consumption overhead for the `DRAM` RAPL domain

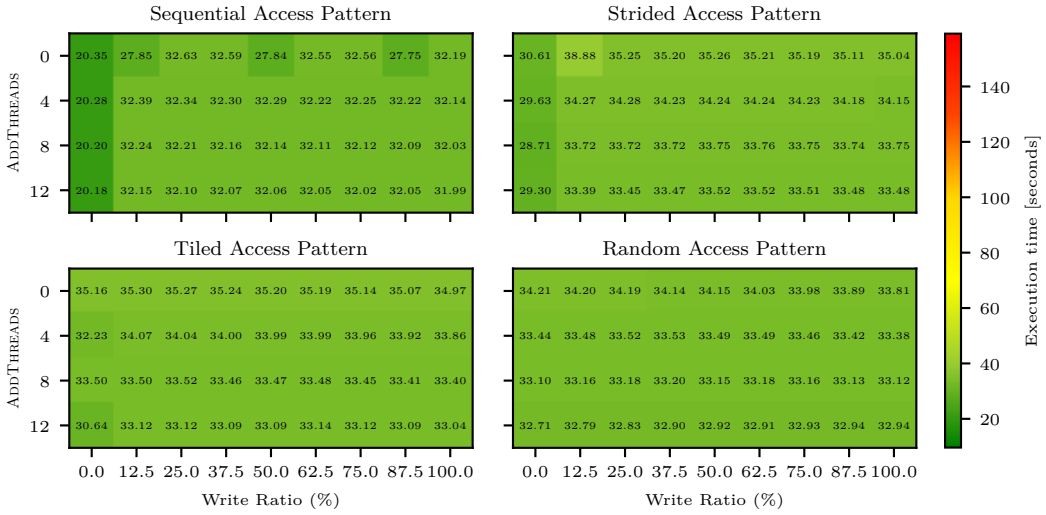


(f) Energy consumption overhead for the `Package` RAPL domain

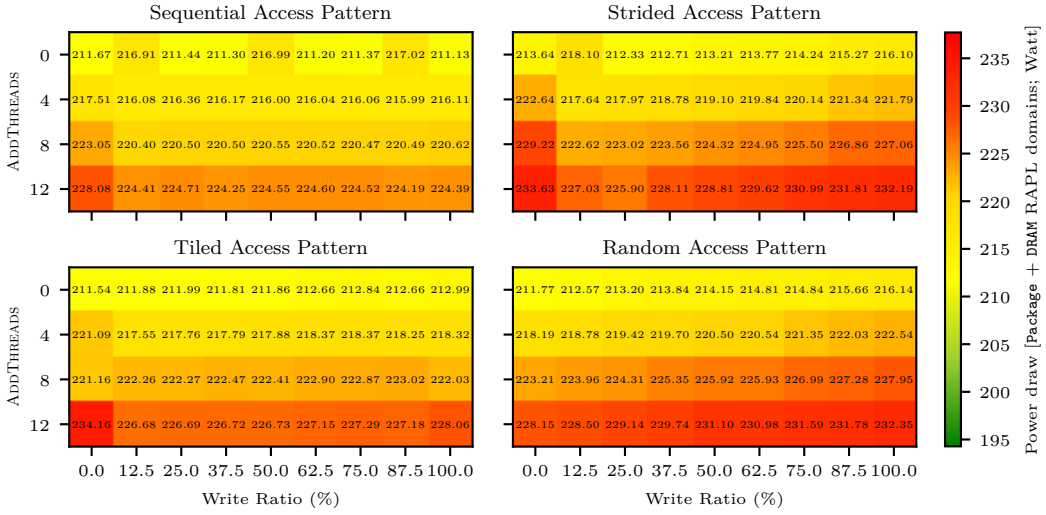
Figure B.8.: Power draw and energy consumption overheads of ProfDP's bandwidth-aware placement decisions over the optimal placements

B.2. NVM Profiling Experiment

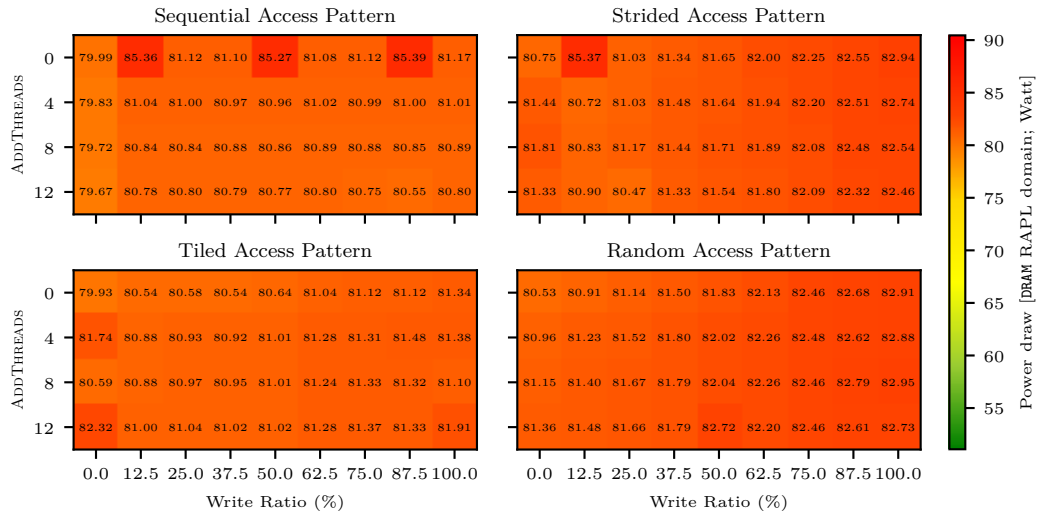
B.2.1. H2M



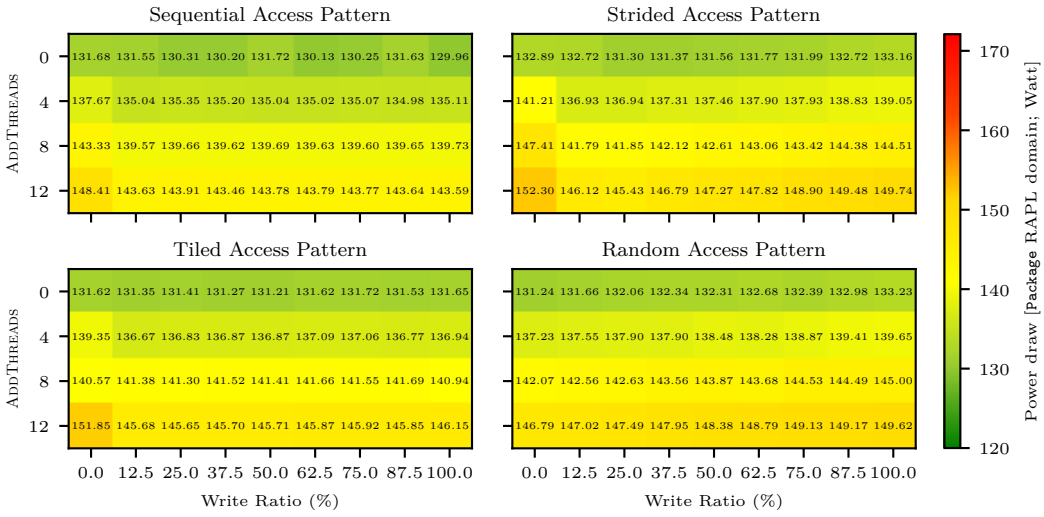
(a) Execution time



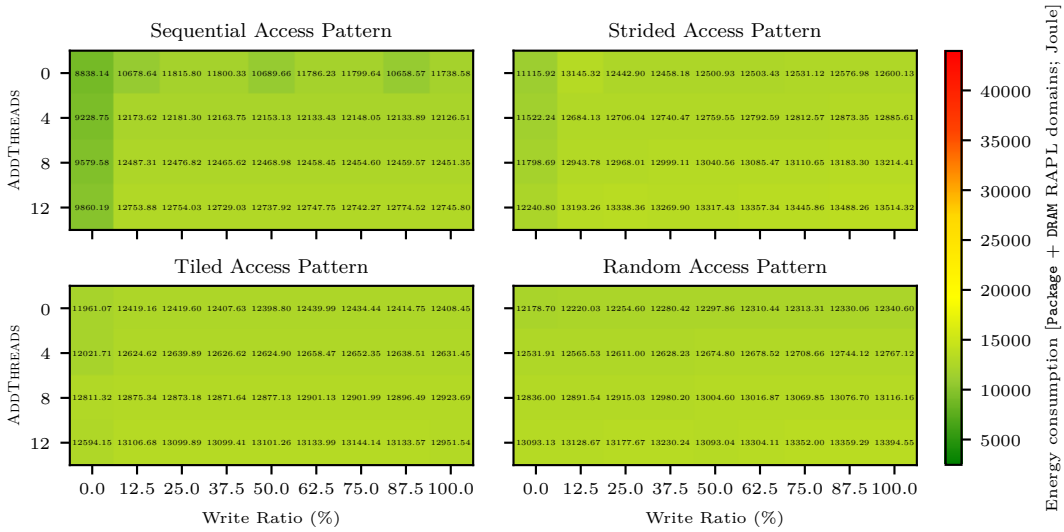
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



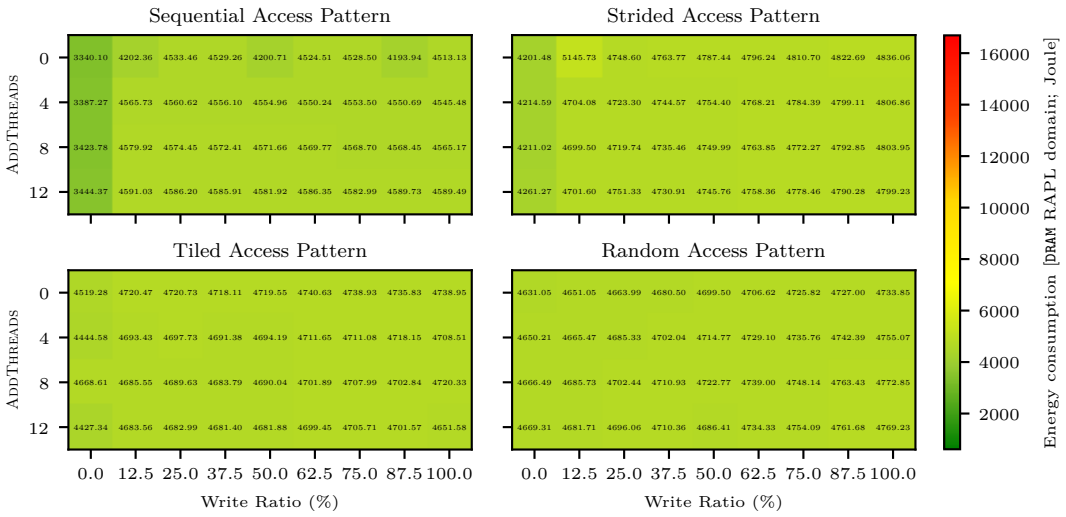
(c) Power draw (DRAM RAPL domain)



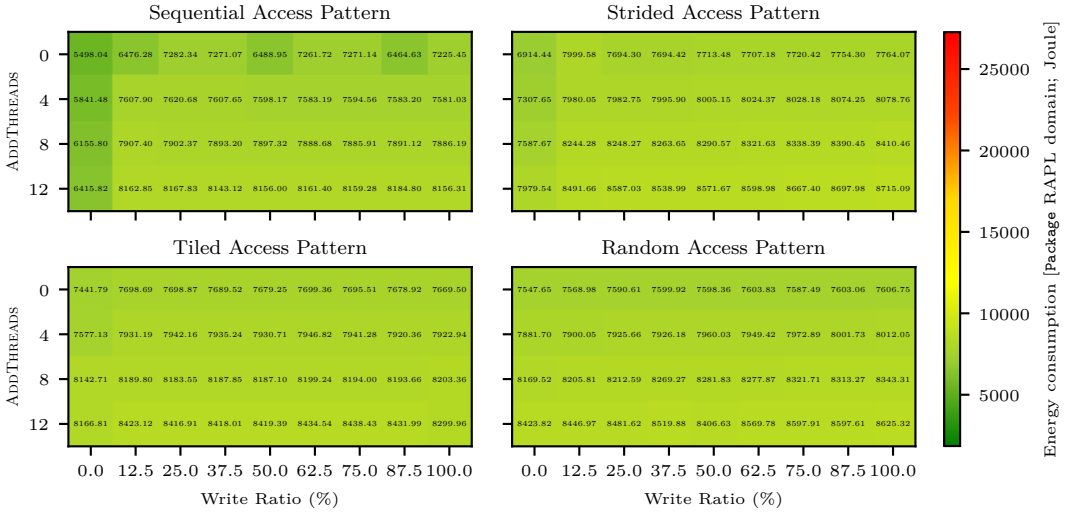
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

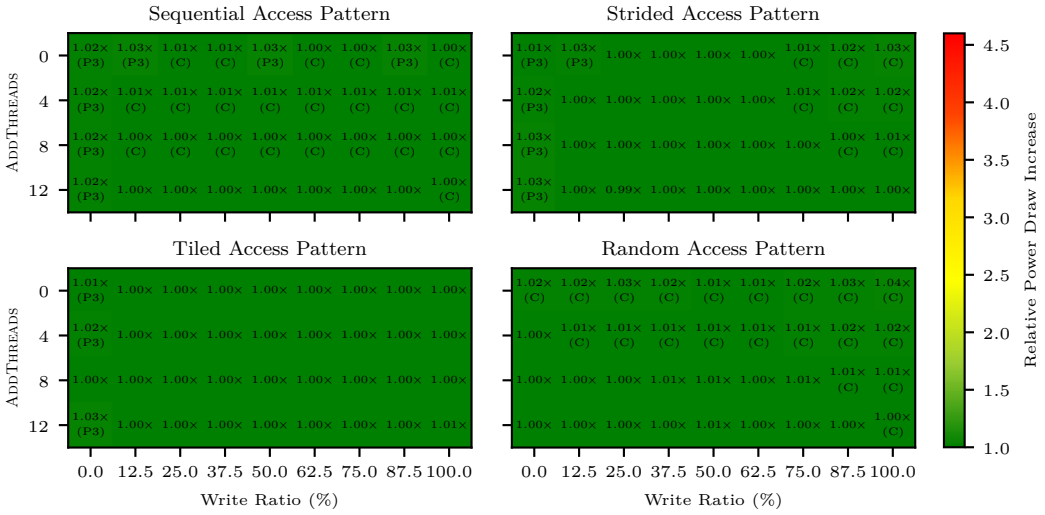


(f) Energy consumption (DRAM RAPL domain)

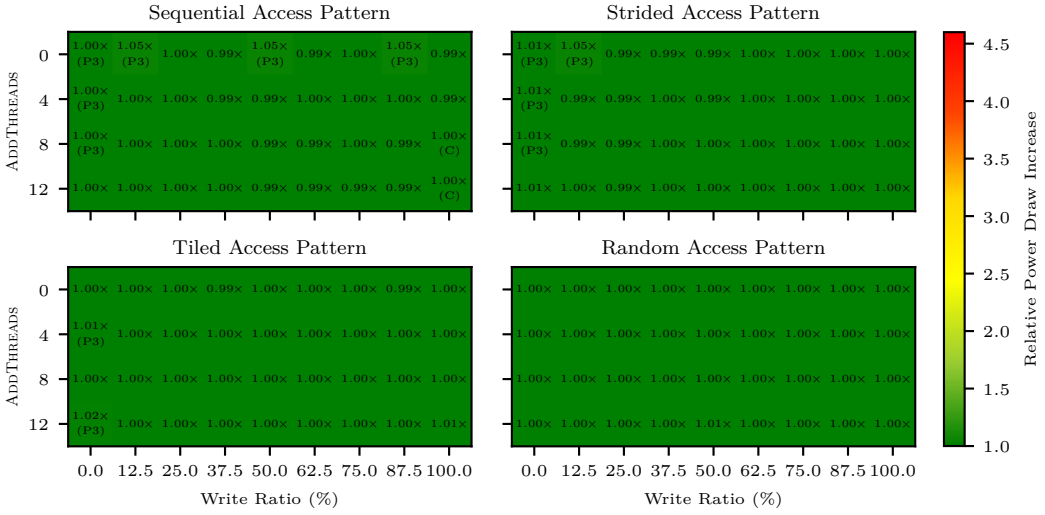


(g) Energy consumption (Package RAPL domain)

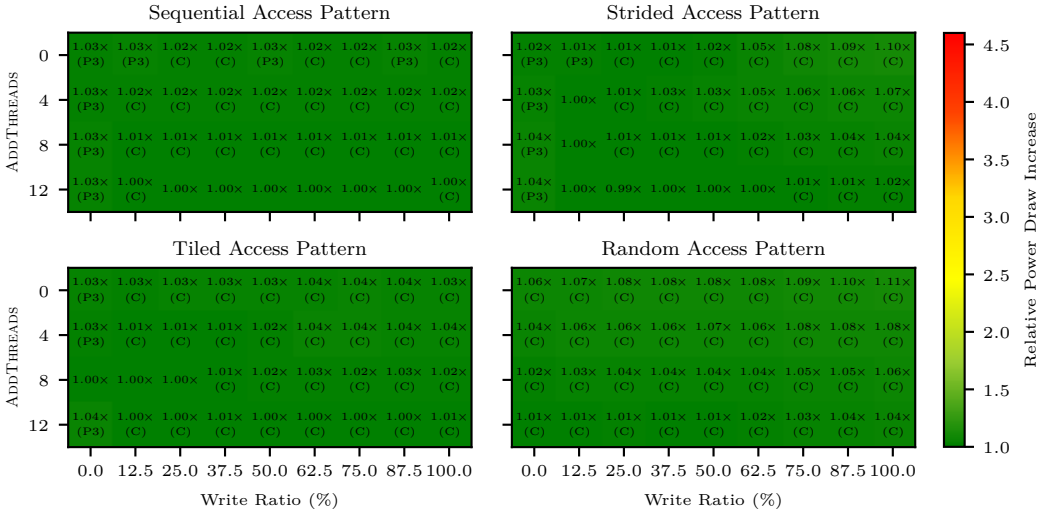
Figure B.9.: Execution time, power draw, and energy consumption of H2M's placement decisions



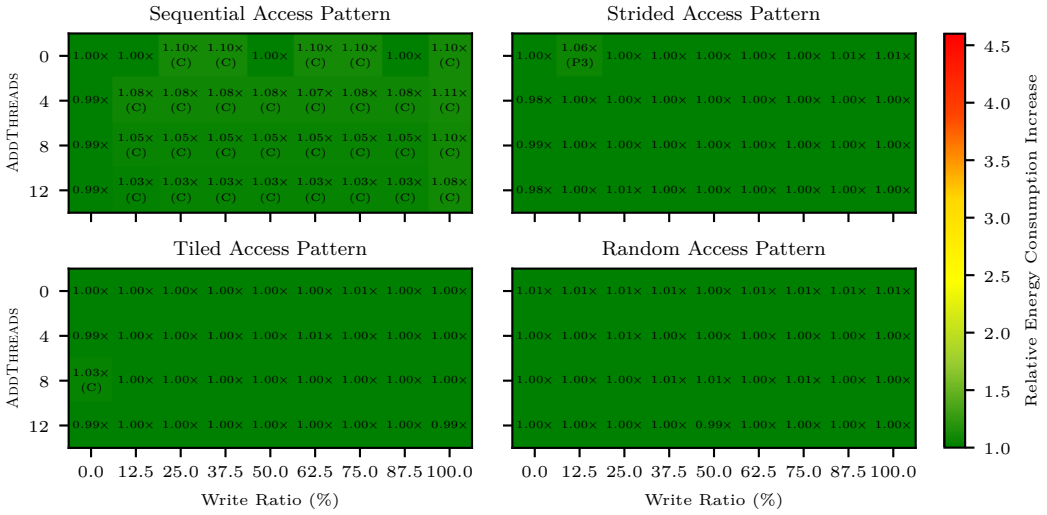
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



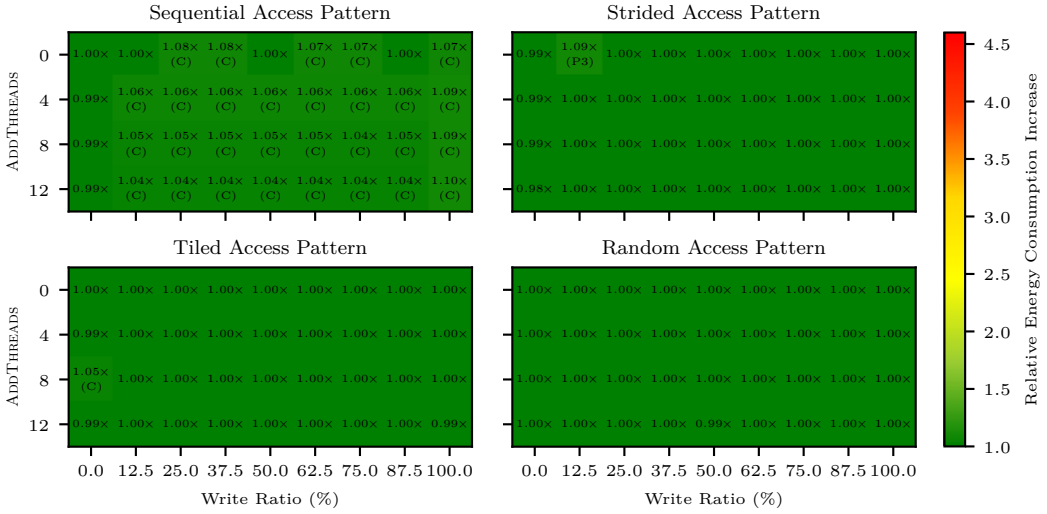
(b) Power draw overhead for the `DRAM` RAPL domain



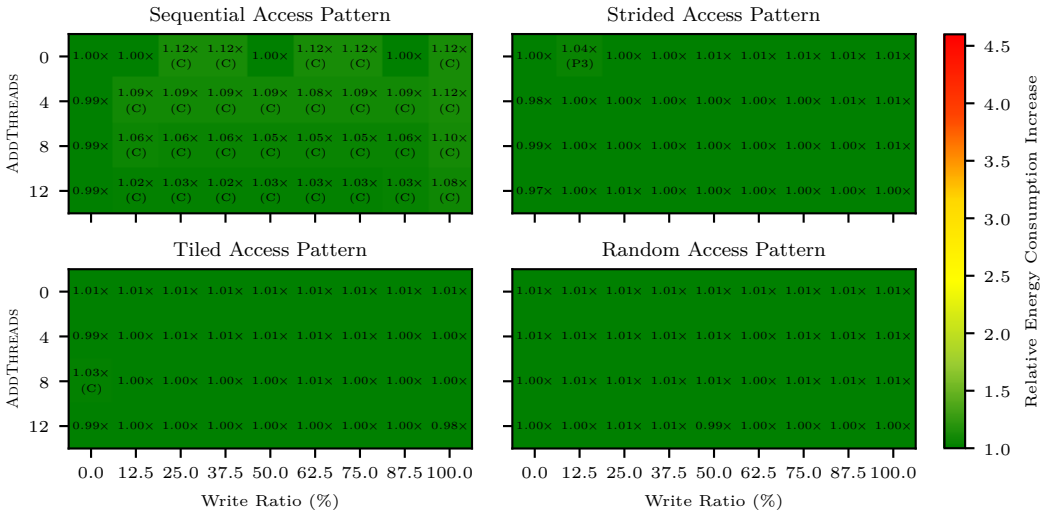
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



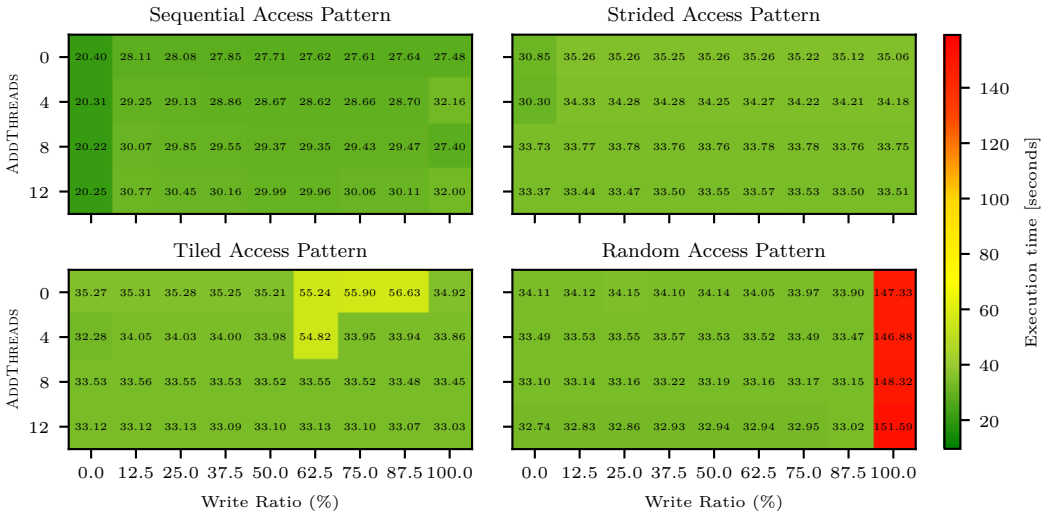
(e) Energy consumption overhead for the `DRAM` RAPL domain



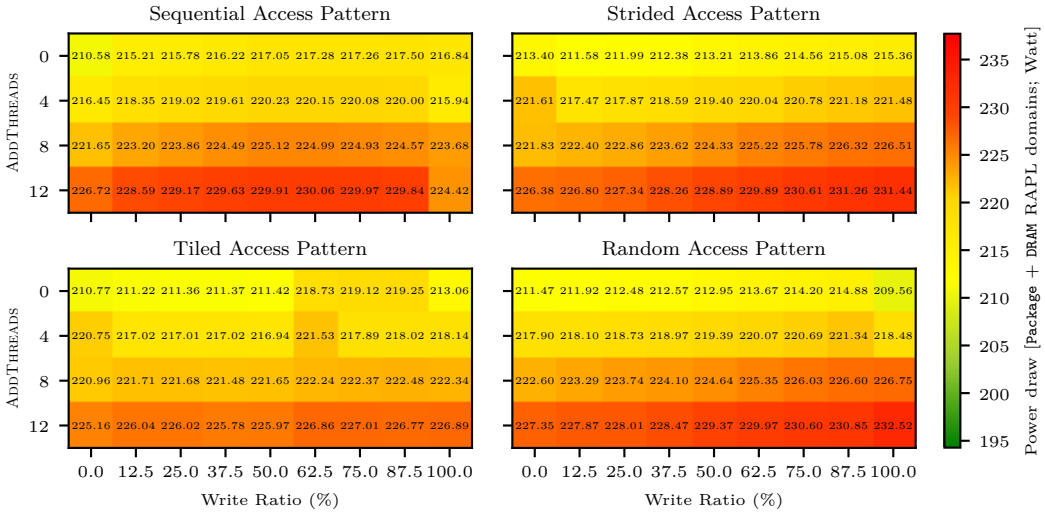
(f) Energy consumption overhead for the `Package` RAPL domain

Figure B.10.: Power draw and energy consumption overheads of H2M's placement decisions over the optimal placements

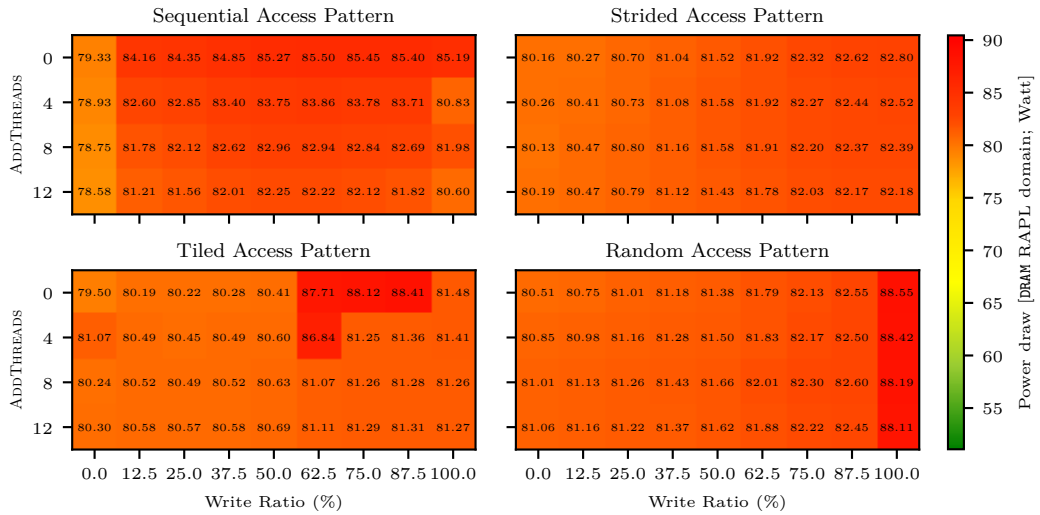
B.2.2. ecoHMEM



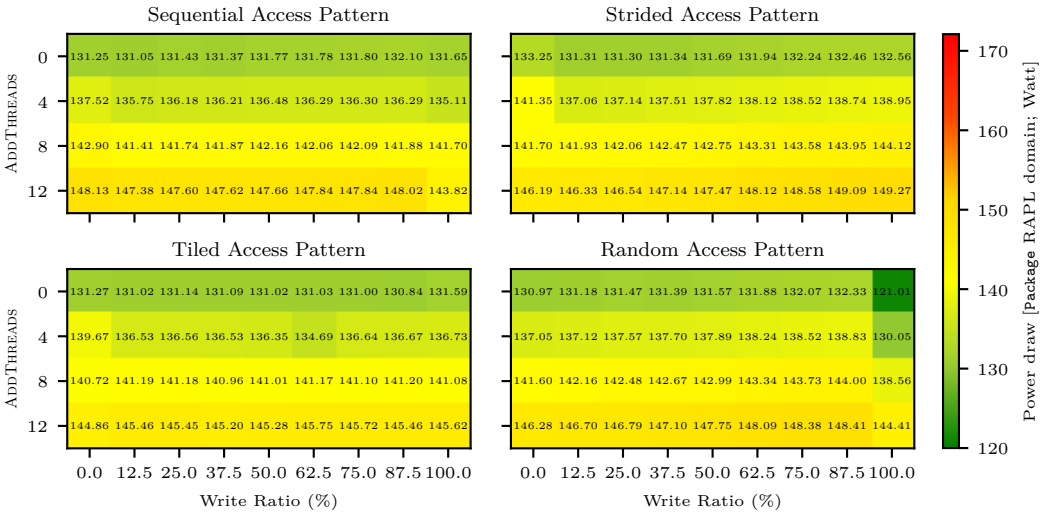
(a) Execution time



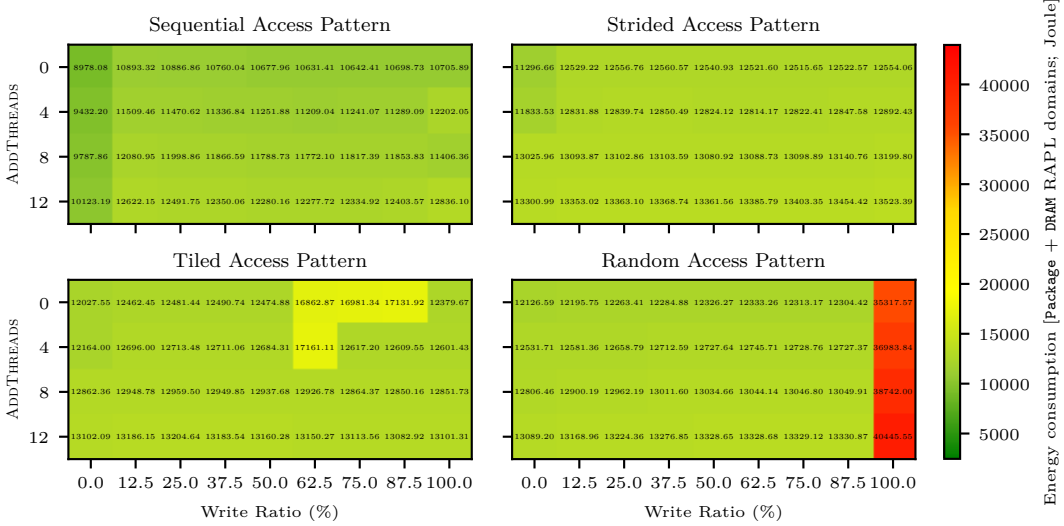
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



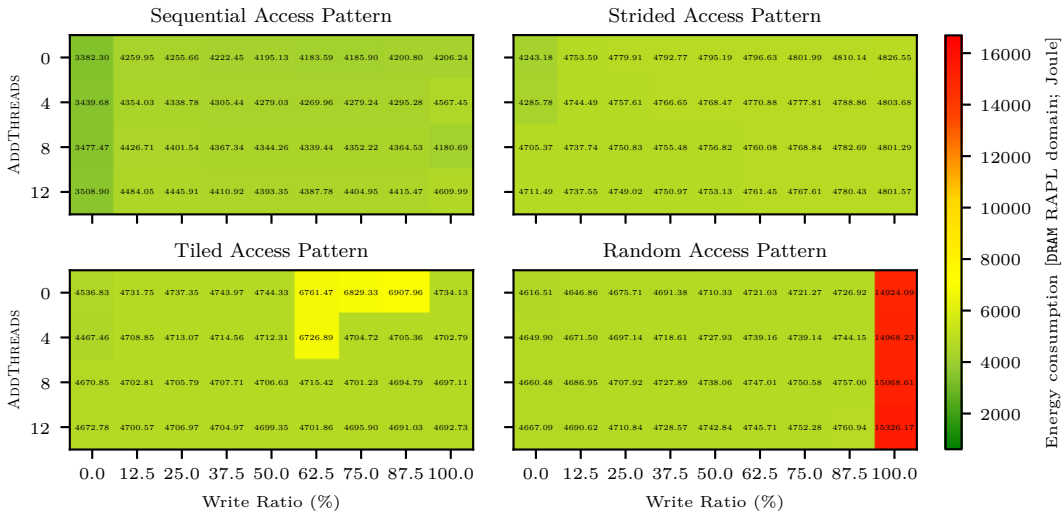
(c) Power draw (DRAM RAPL domain)



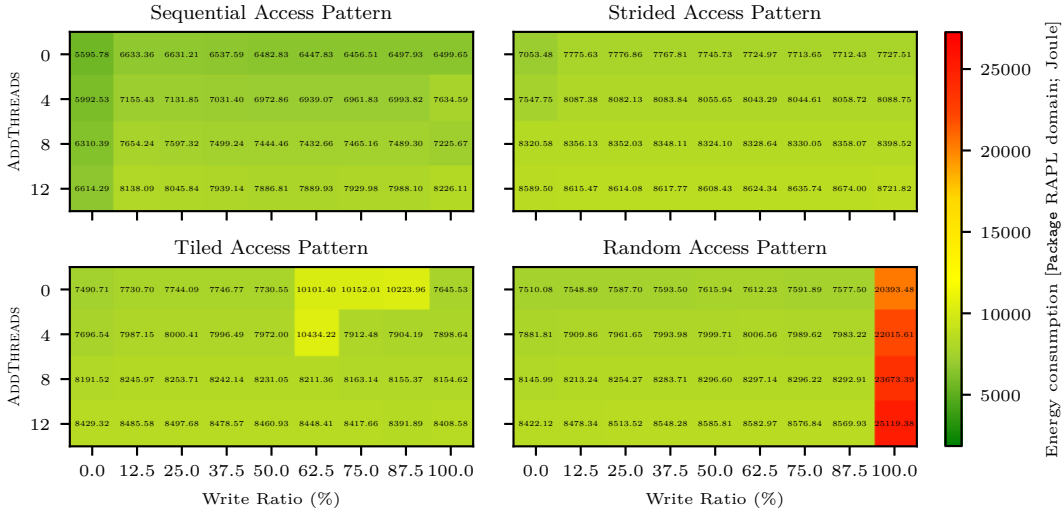
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

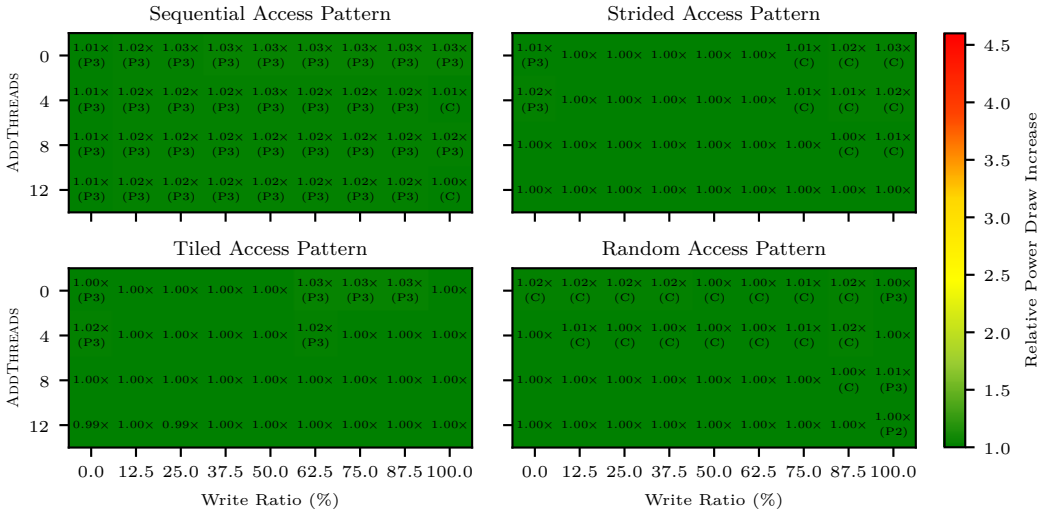


(f) Energy consumption (`DRAM` RAPL domain)

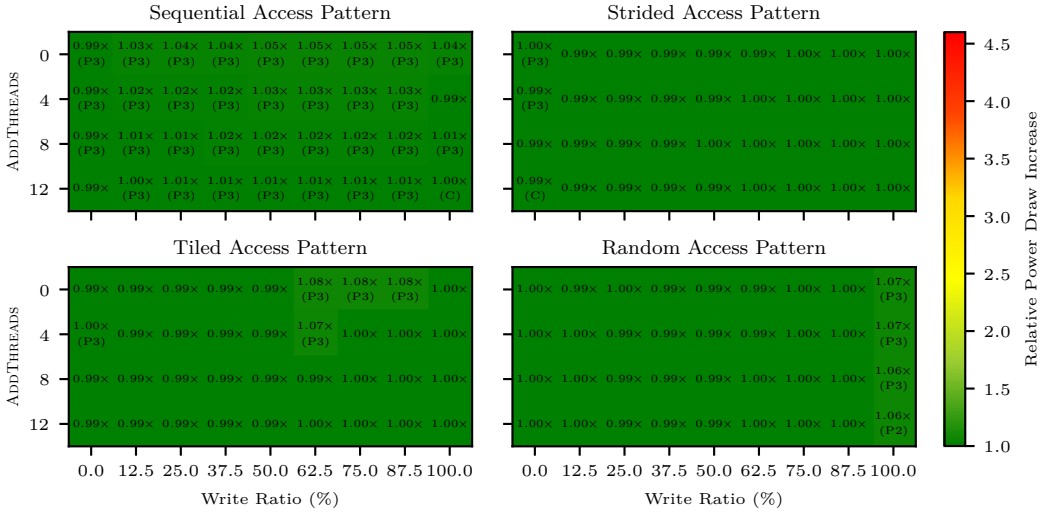


(g) Energy consumption (`Package` RAPL domain)

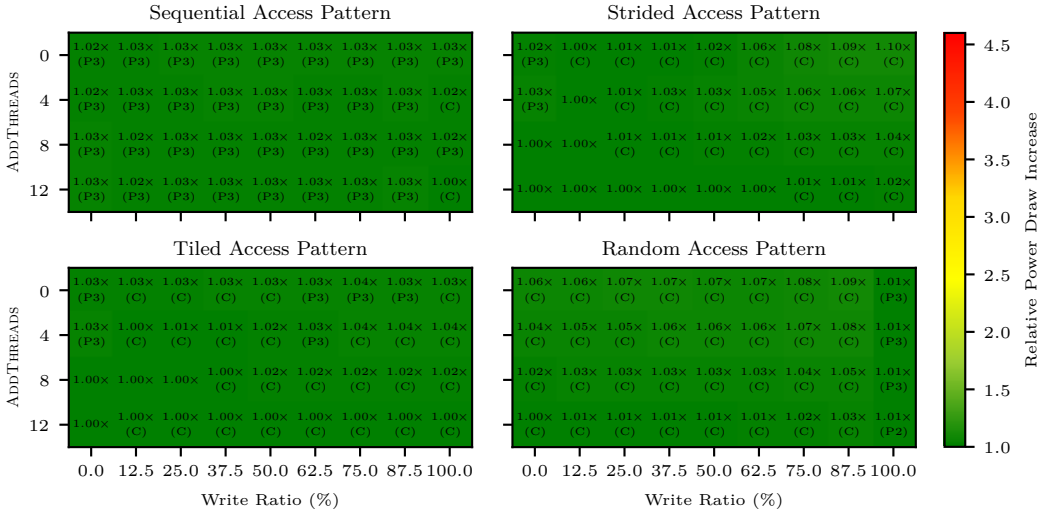
Figure B.11.: Execution time, power draw, and energy consumption of ecoHMEM's placement decisions



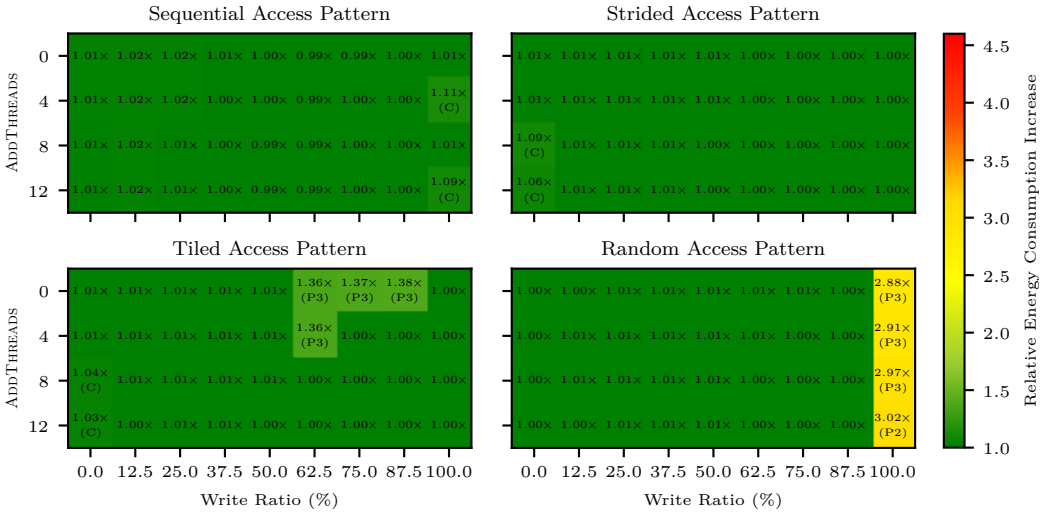
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



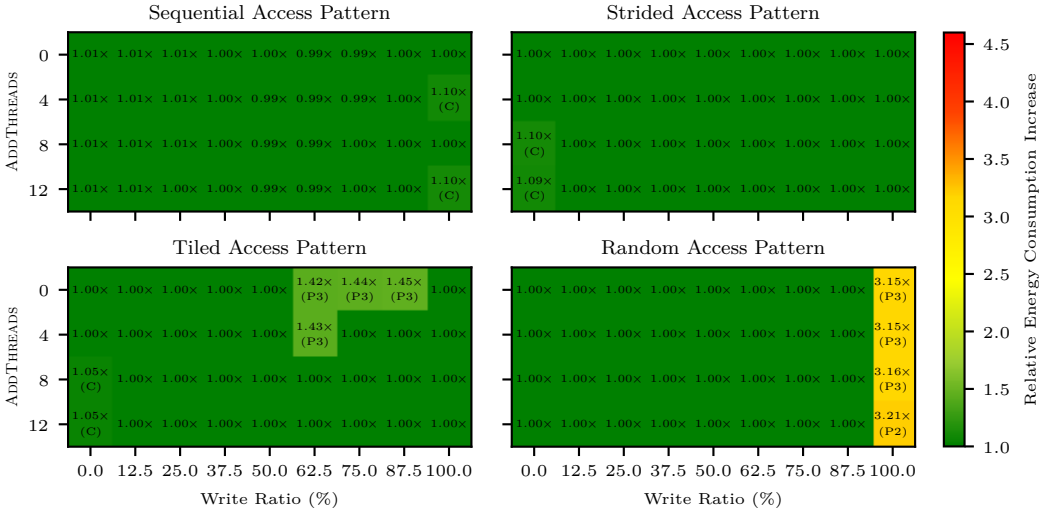
(b) Power draw overhead for the `DRAM` RAPL domain



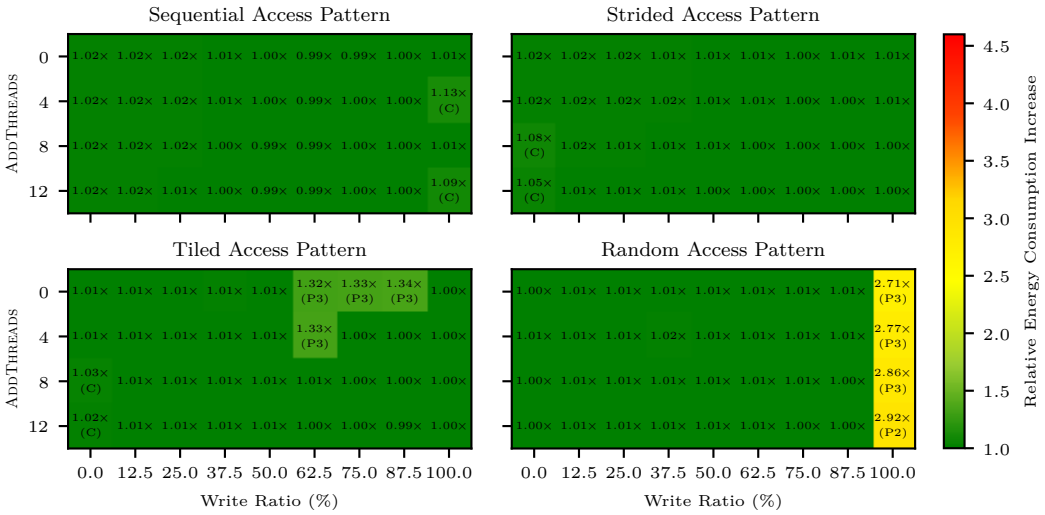
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



(e) Energy consumption overhead for the `DRAM` RAPL domain

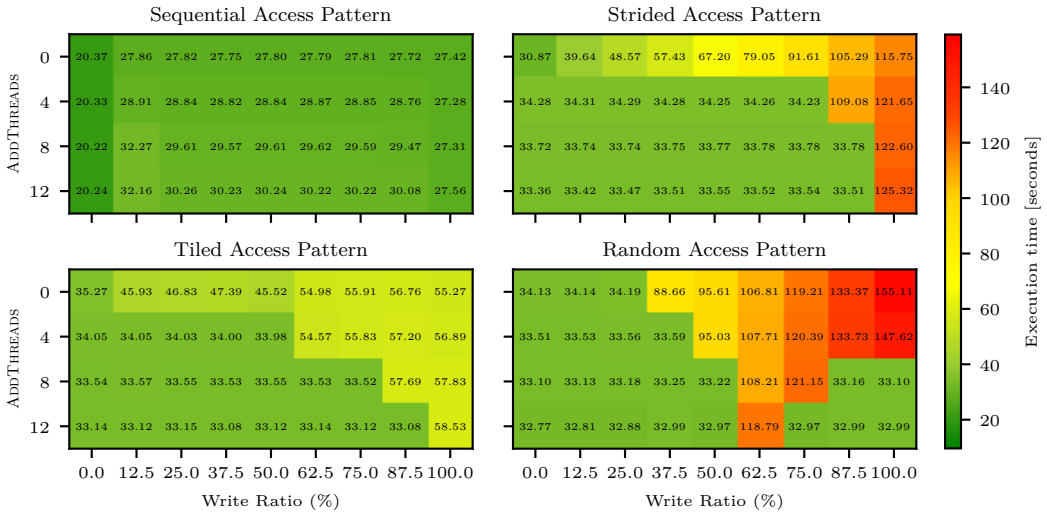


(f) Energy consumption overhead for the `Package` RAPL domain

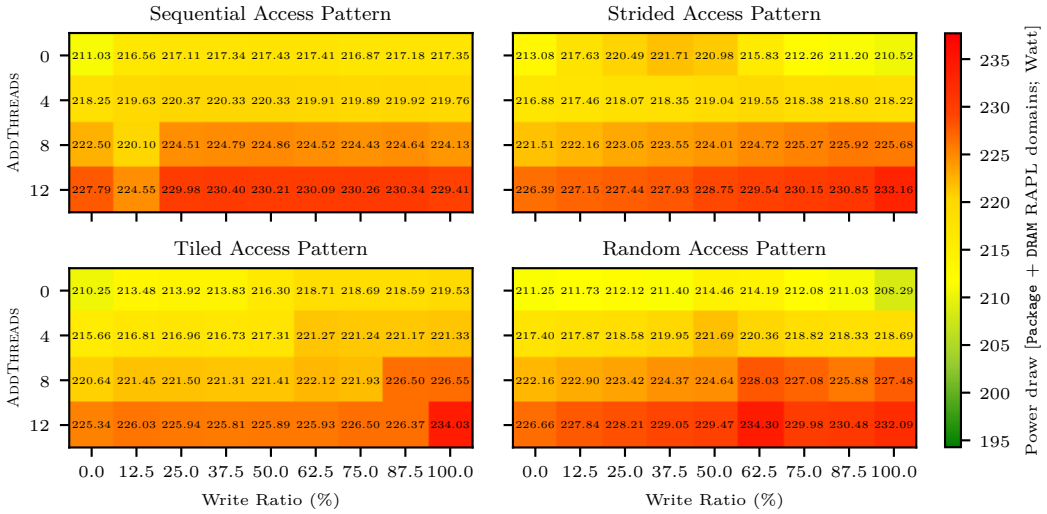
Figure B.12.: Power draw and energy consumption overheads of ecoHMEM's placement decisions over the optimal placements

B.2.3. ProfDP

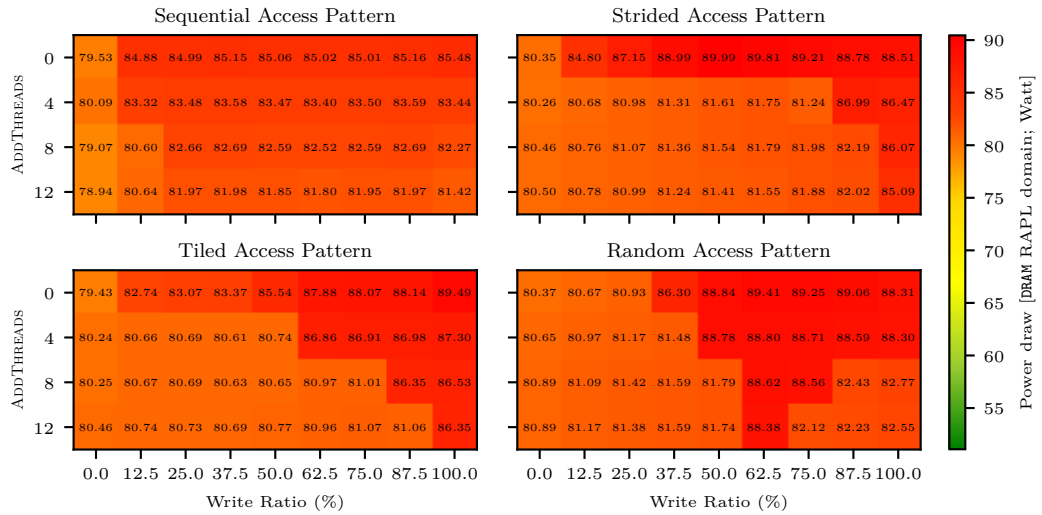
Latency-Aware Placement Optimization



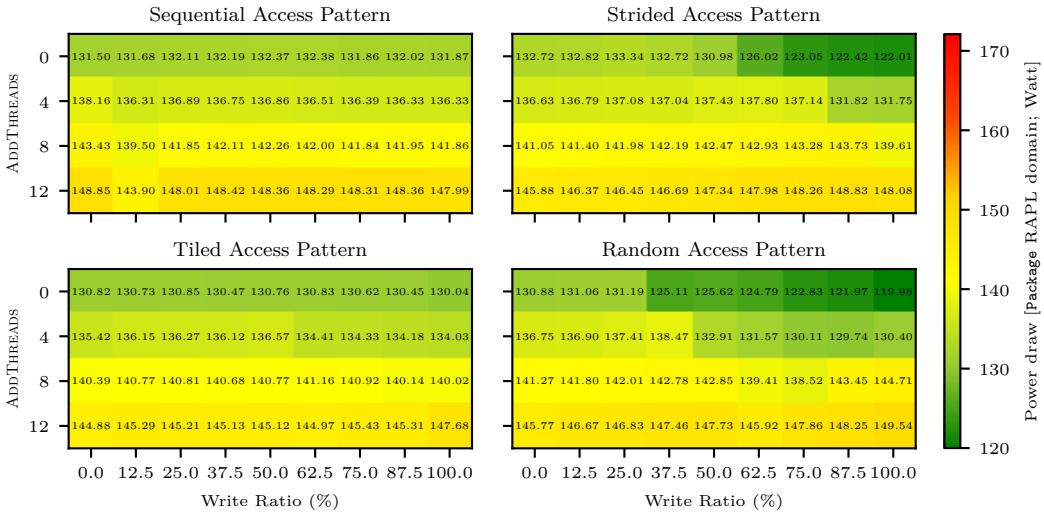
(a) Execution time



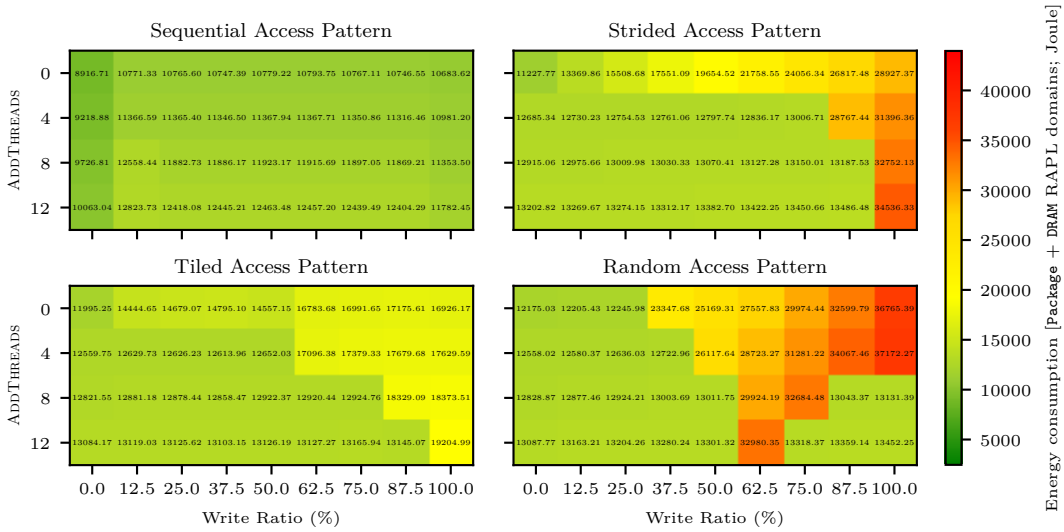
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



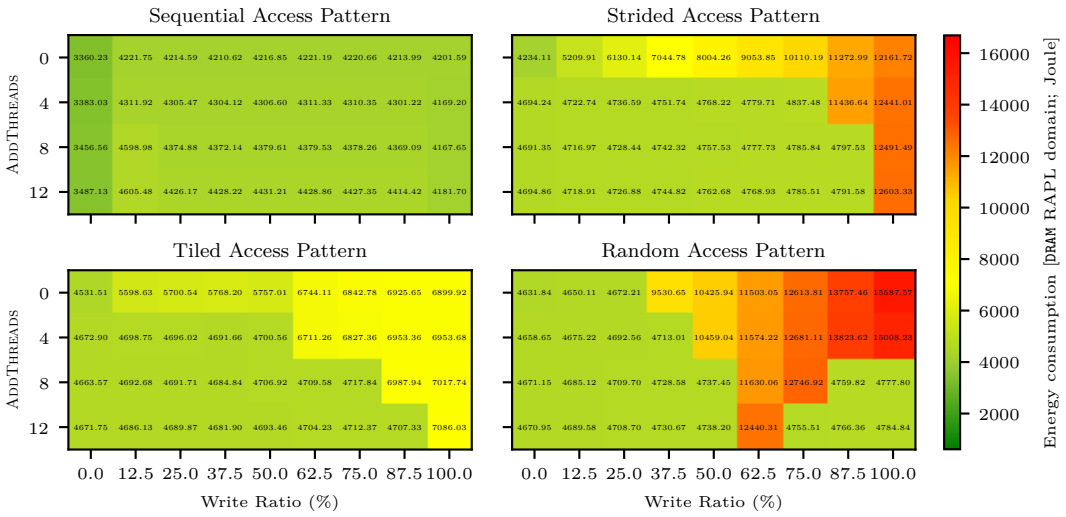
(c) Power draw (DRAM RAPL domain)



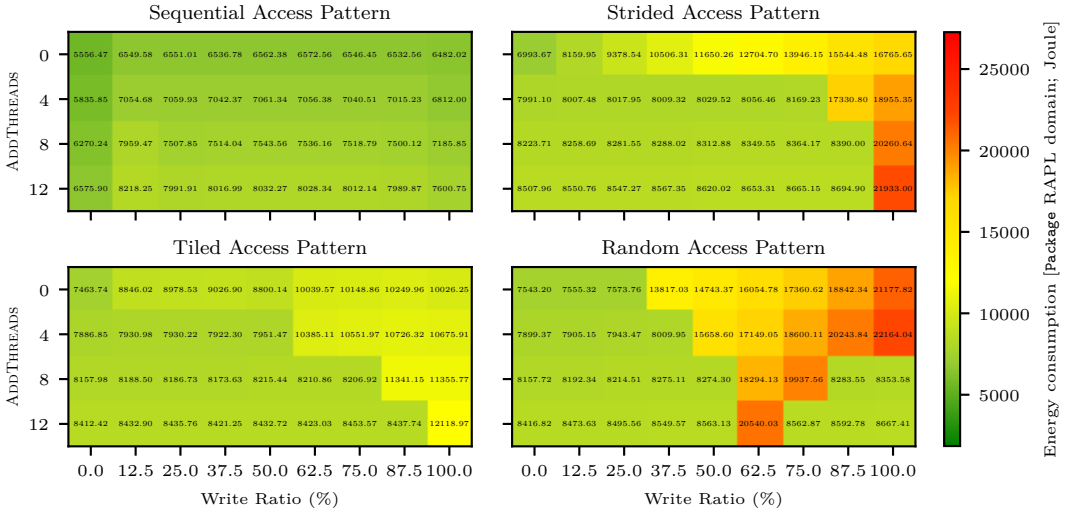
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

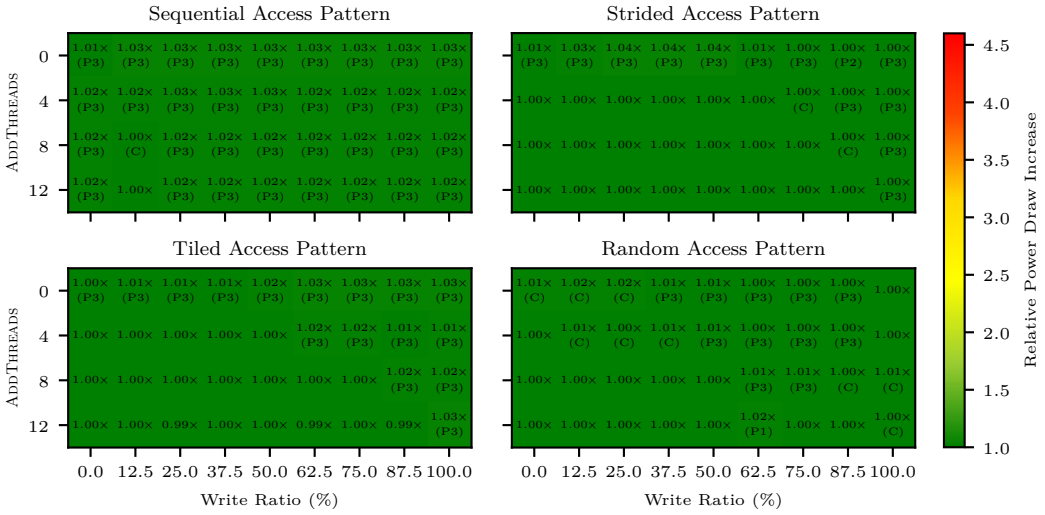


(f) Energy consumption (DRAM RAPL domain)

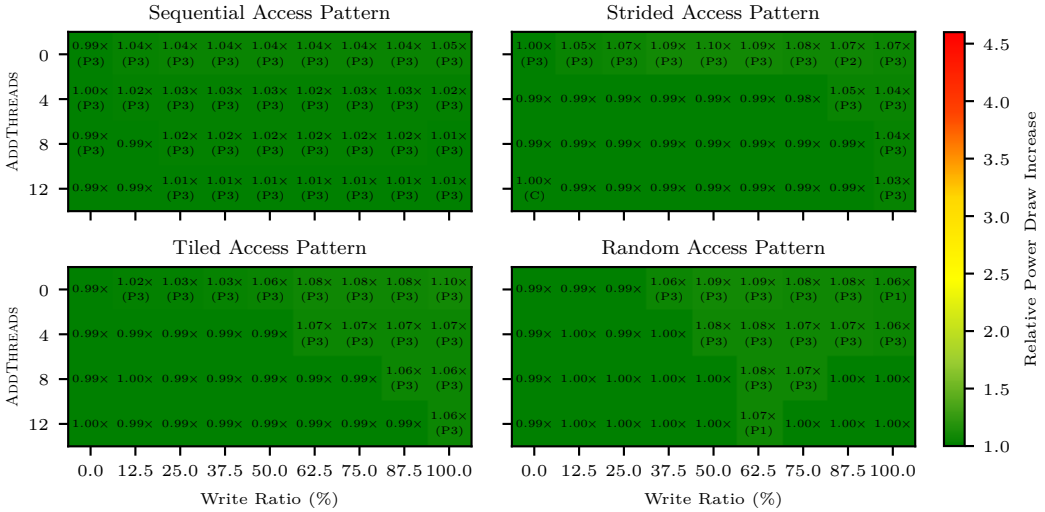


(g) Energy consumption (Package RAPL domain)

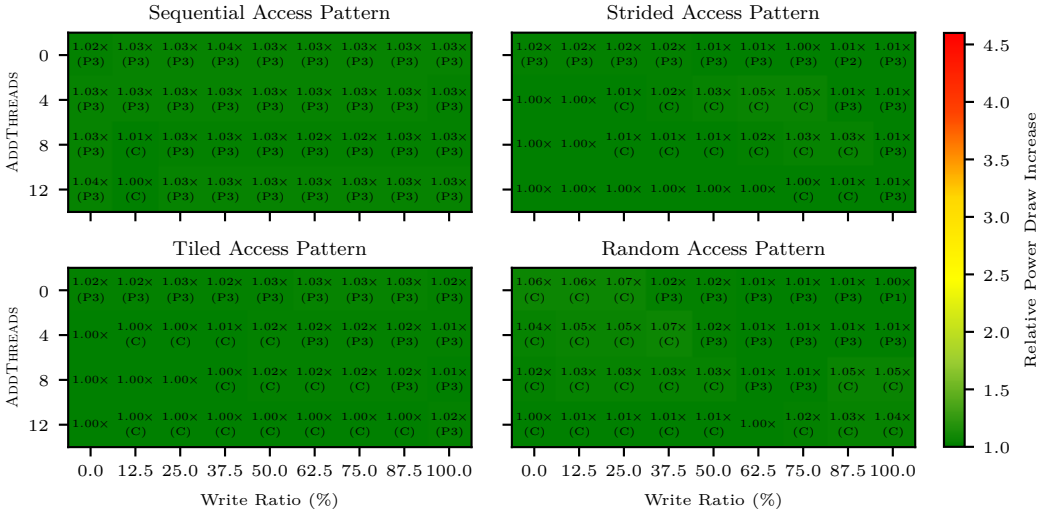
Figure B.13.: Execution time, power draw, and energy consumption of ProfDP's latency-aware placement decisions



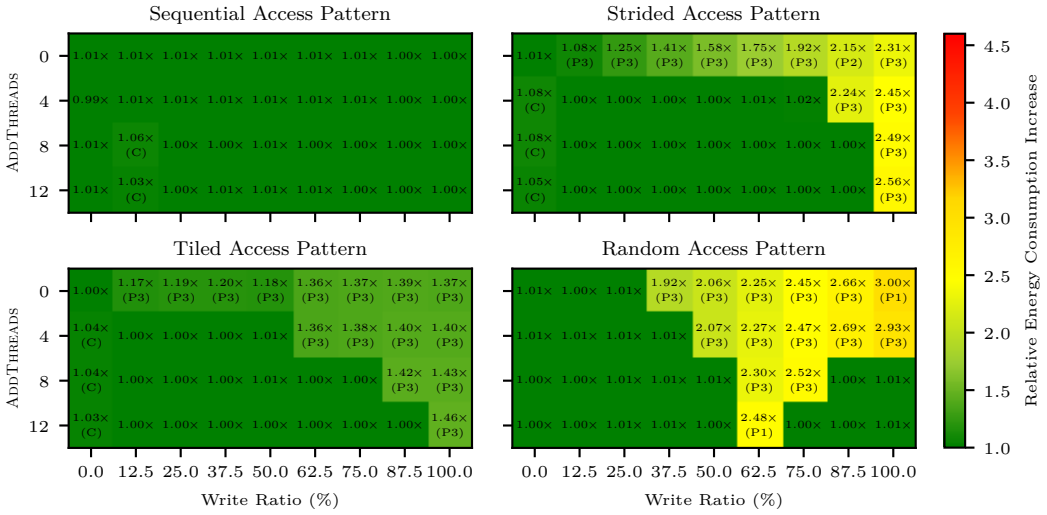
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



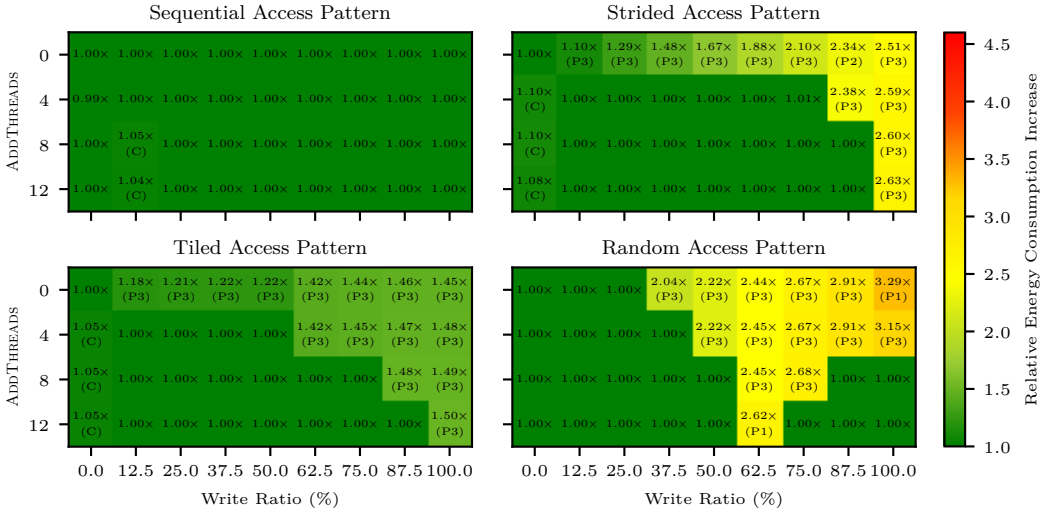
(b) Power draw overhead for the `DRAM` RAPL domain



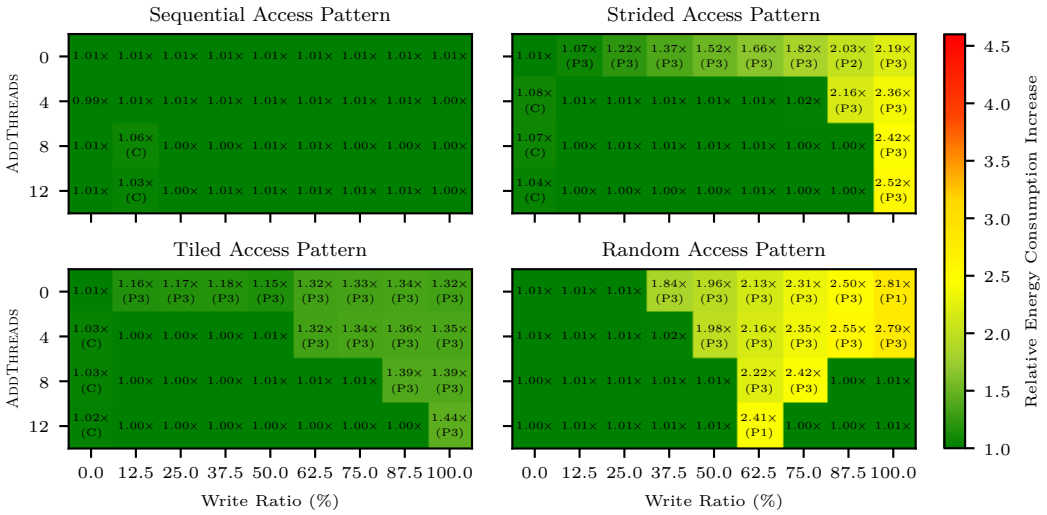
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



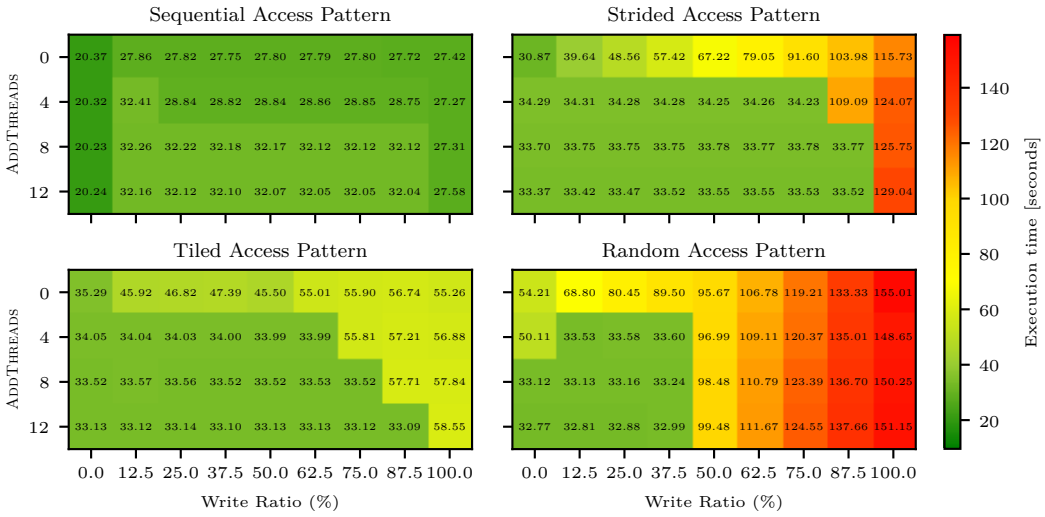
(e) Energy consumption overhead for the `DRAM` RAPL domain



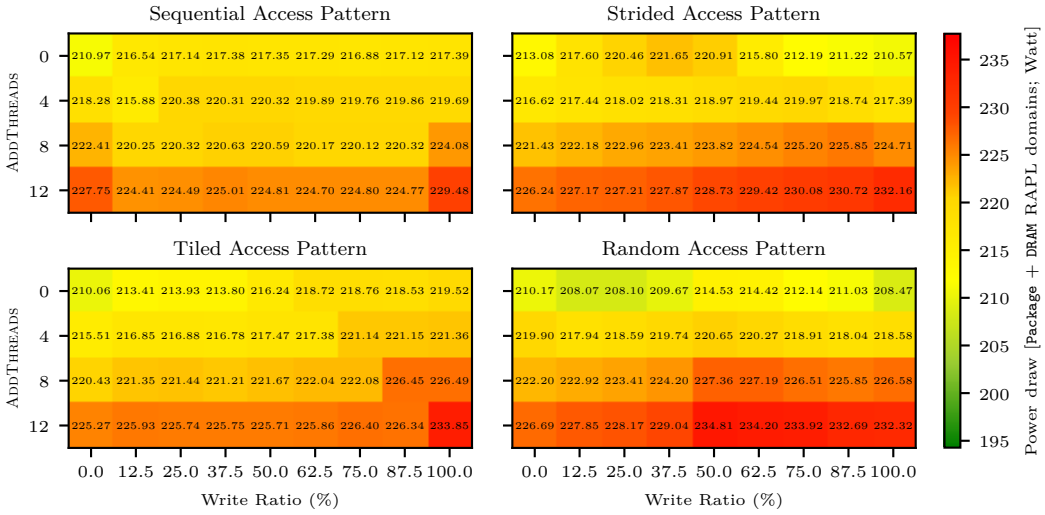
(f) Energy consumption overhead for the `Package` RAPL domain

Figure B.14.: Power draw and energy consumption overheads of ProfDP's latency-aware placement decisions over the optimal placements

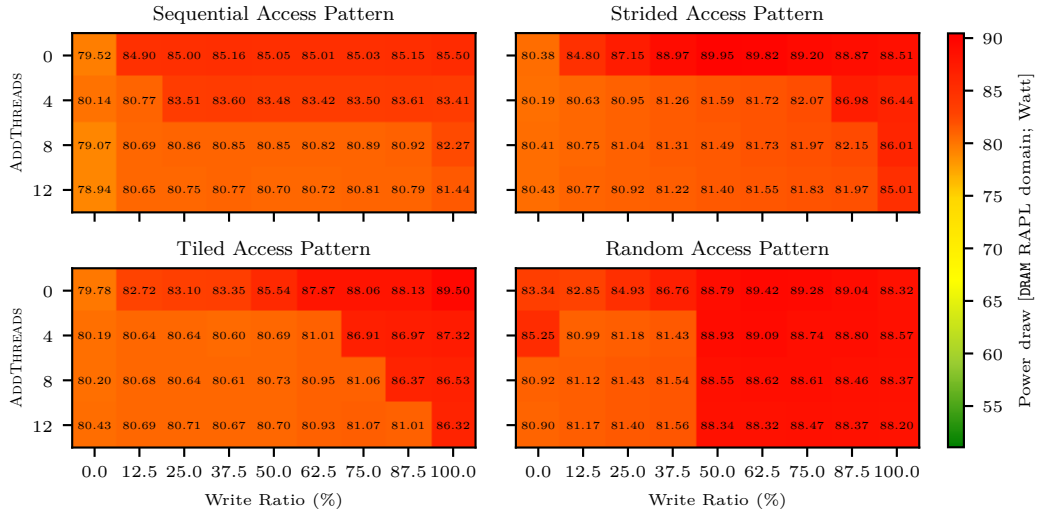
Bandwidth-Aware Placement Optimization



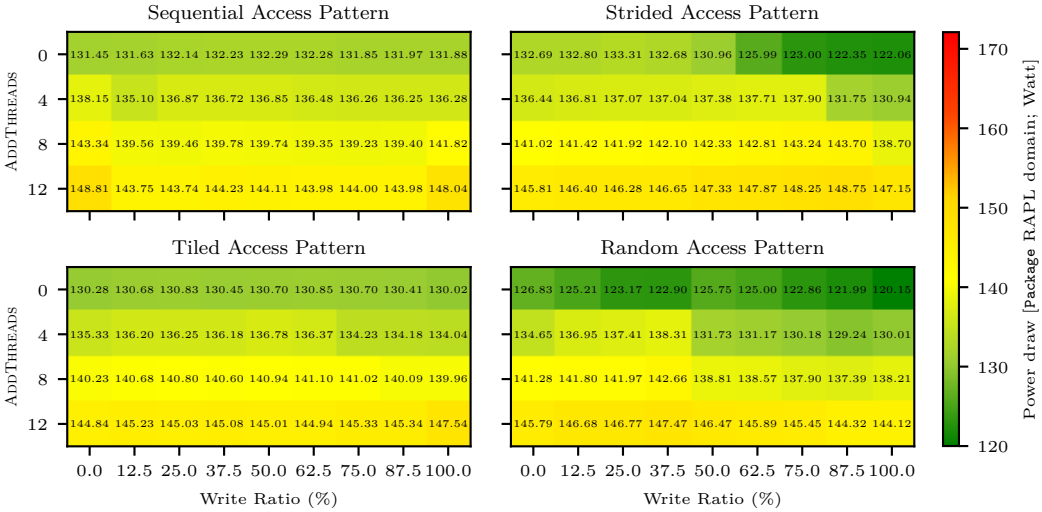
(a) Execution time



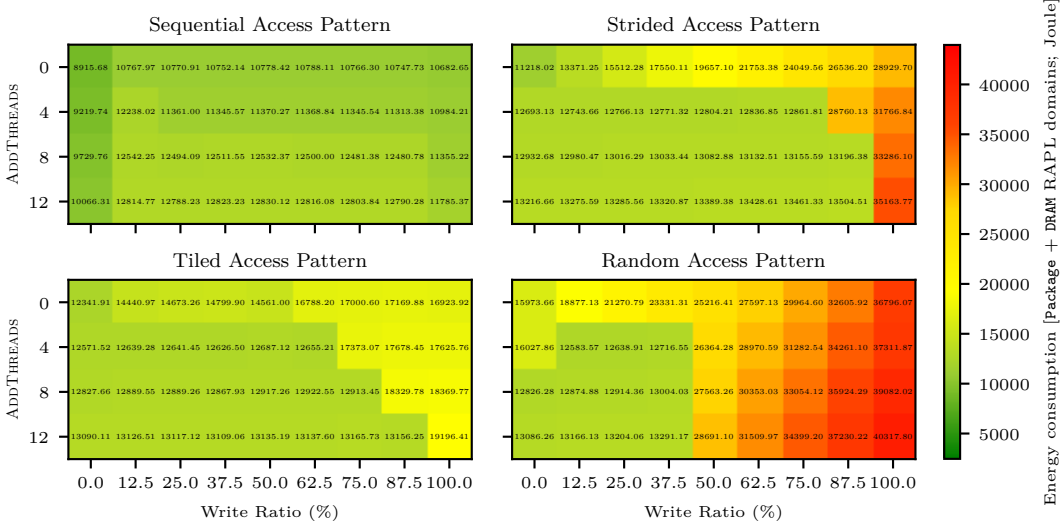
(b) Power draw (`Package` and `DRAM` RAPL domains combined)



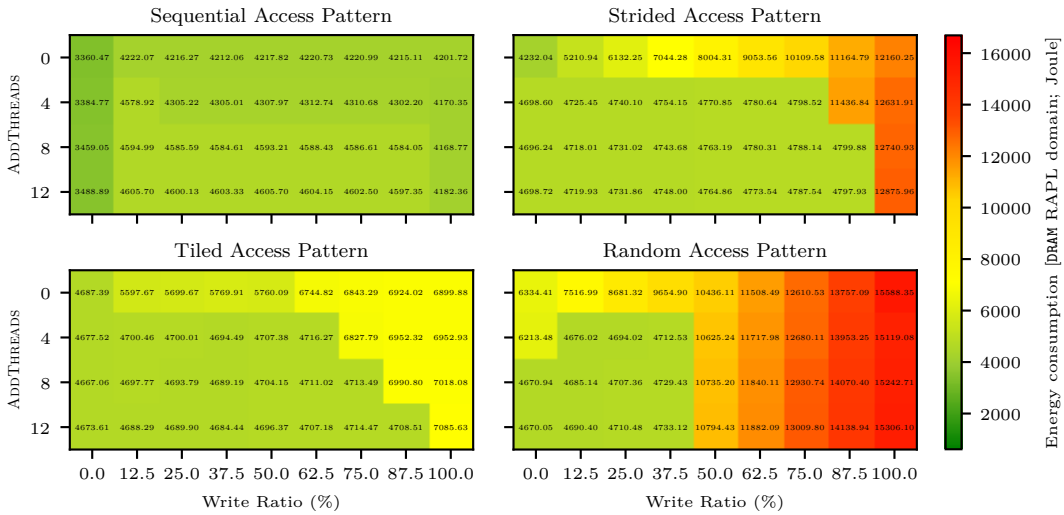
(c) Power draw (DRAM RAPL domain)



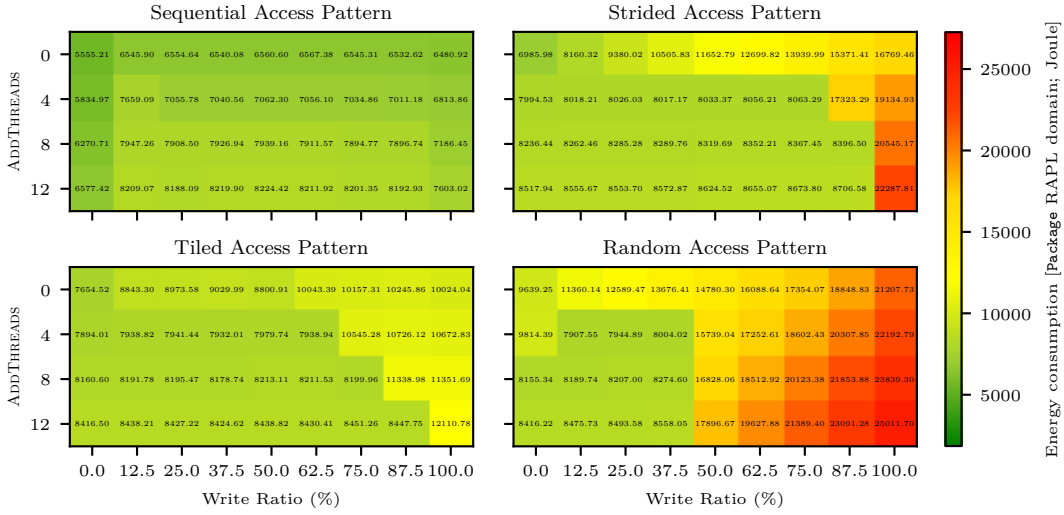
(d) Power draw (Package RAPL domain)



(e) Energy consumption (Package and DRAM RAPL domains combined)

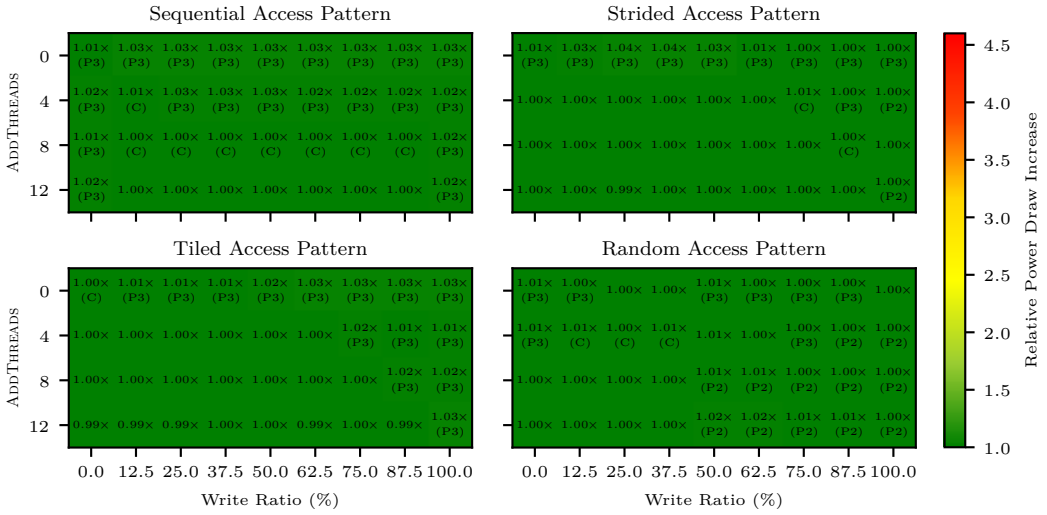


(f) Energy consumption (DRAM RAPL domain)

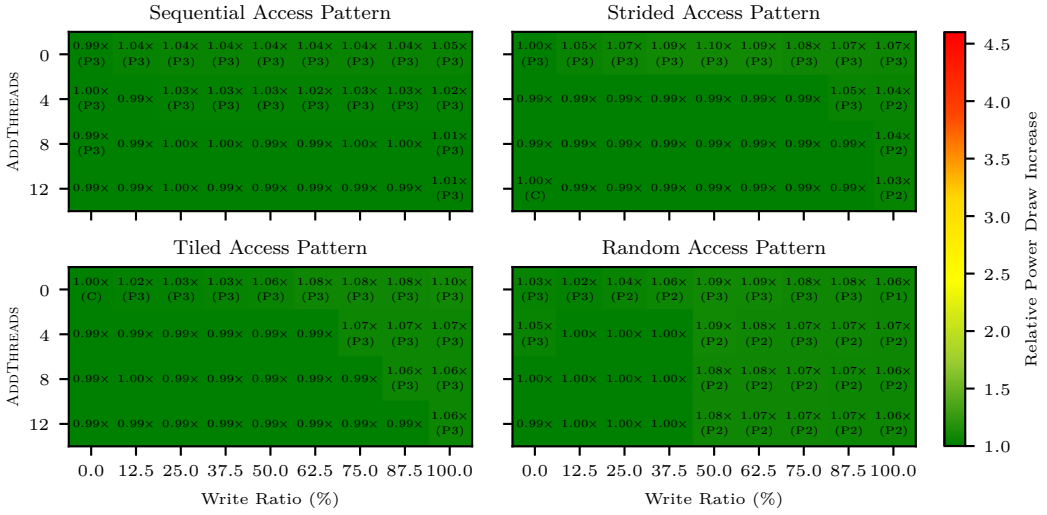


(g) Energy consumption (Package RAPL domain)

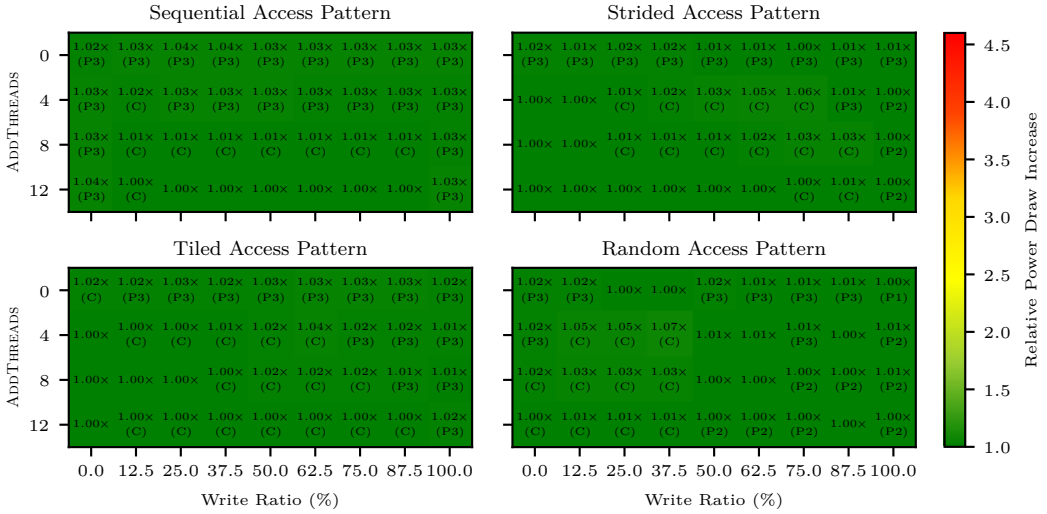
Figure B.15.: Execution time, power draw, and energy consumption of ProfDP's bandwidth-aware placement decisions



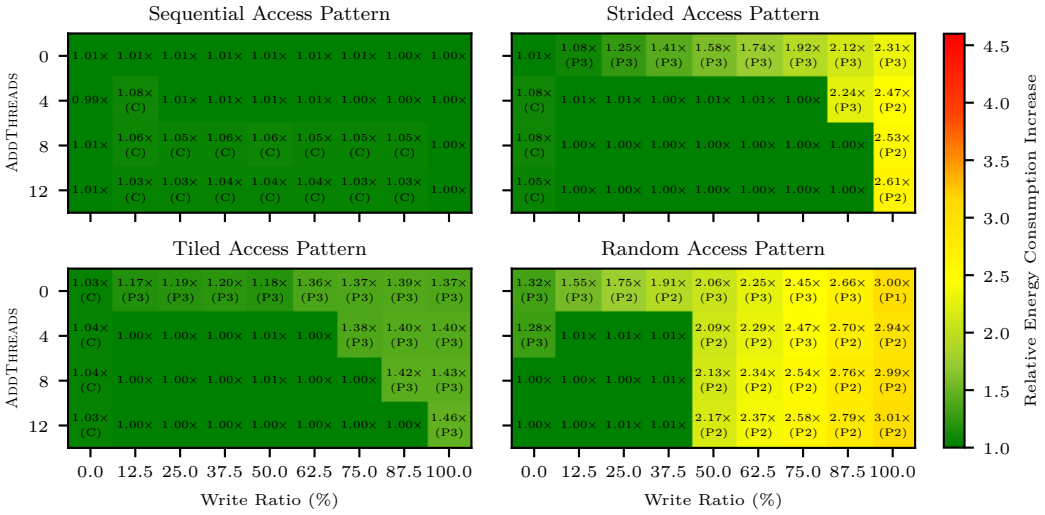
(a) Power draw overhead for the `Package` and `DRAM` RAPL domains



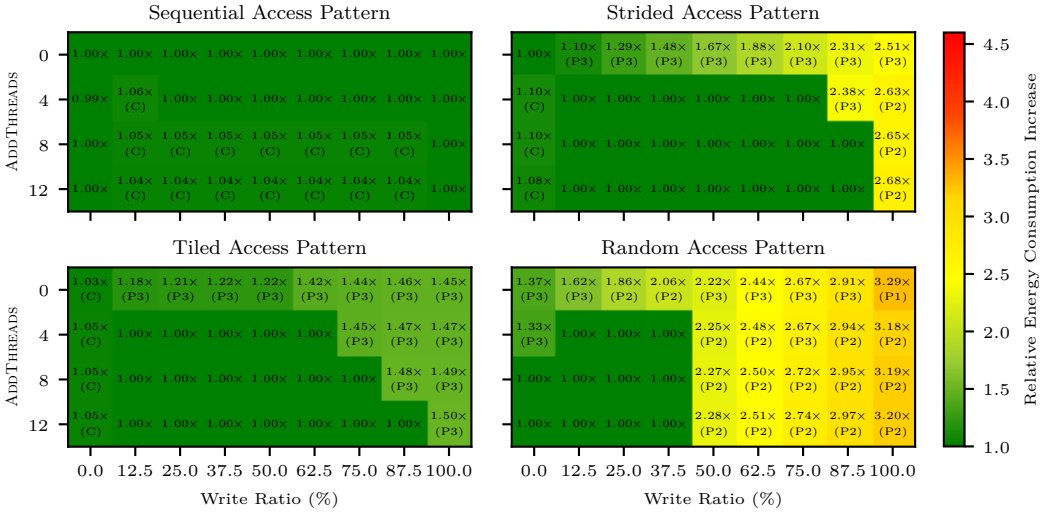
(b) Power draw overhead for the `DRAM` RAPL domain



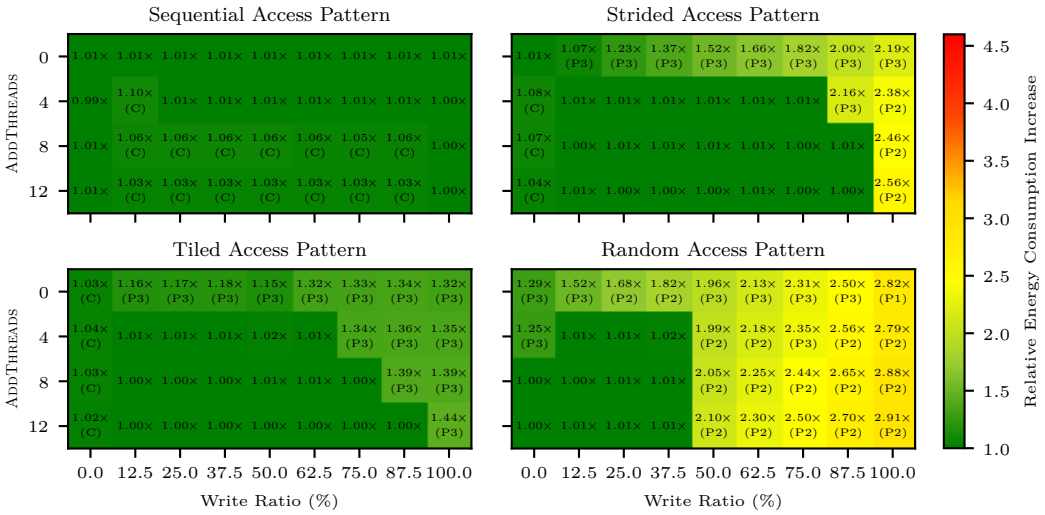
(c) Power draw overhead for the `Package` RAPL domain



(d) Energy consumption overhead for the `Package` and `DRAM` RAPL domains



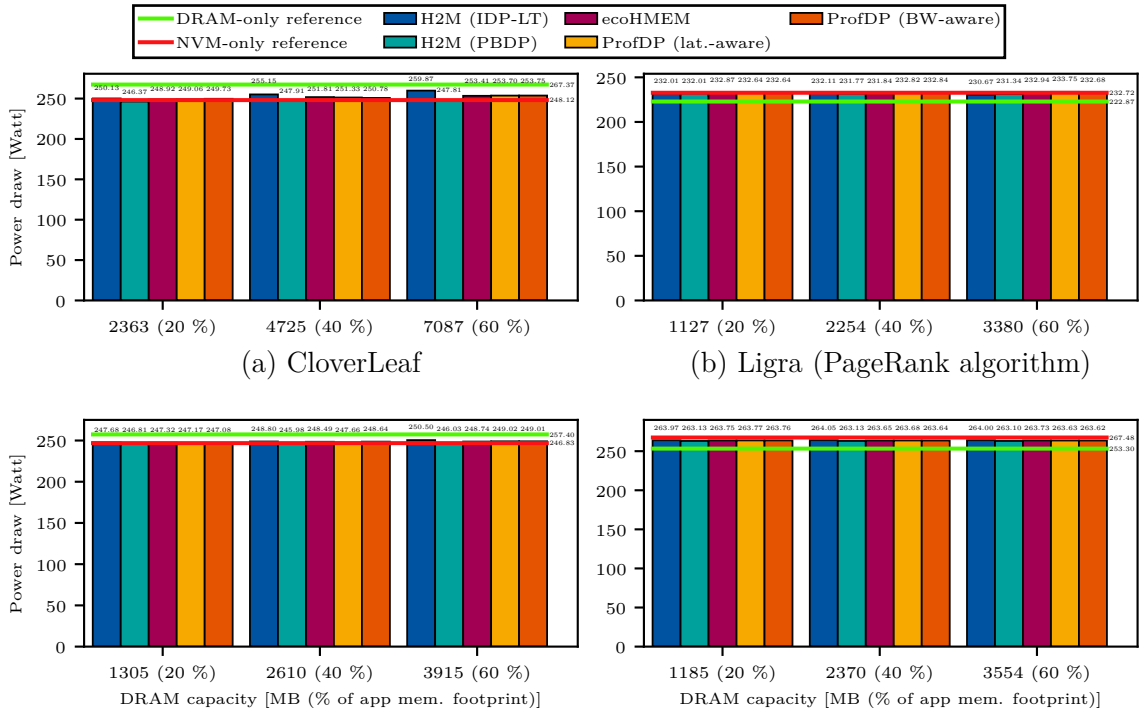
(e) Energy consumption overhead for the `DRAM` RAPL domain



(f) Energy consumption overhead for the `Package` RAPL domain

Figure B.16.: Power draw and energy consumption overheads of ProfDP's bandwidth-aware placement decisions over the optimal placements

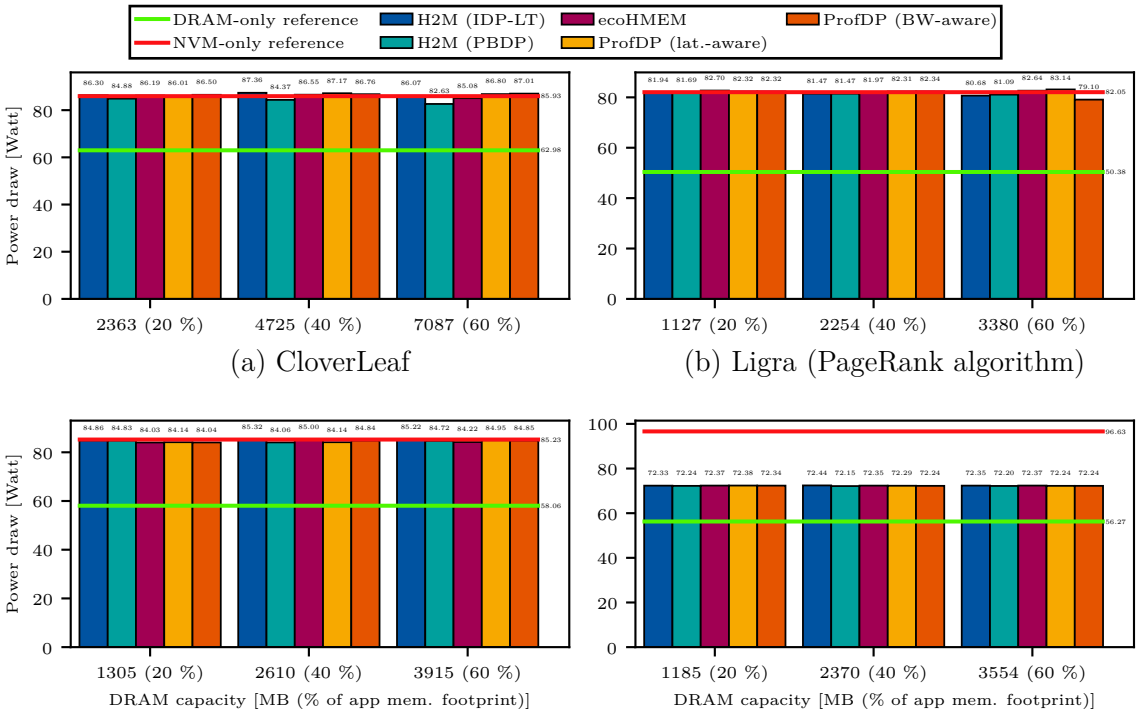
C. Results for the Proxy Applications



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

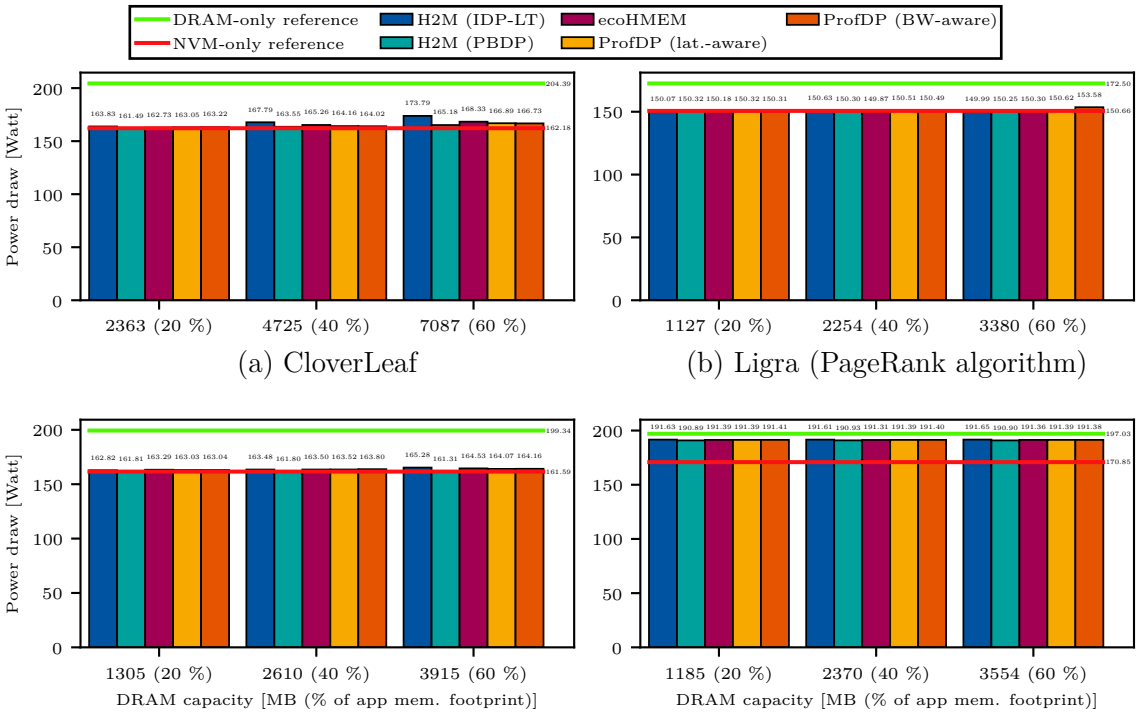
Figure C.1.: Power draw for the proxy applications (`Package` and `DRAM` RAPL domains combined)



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

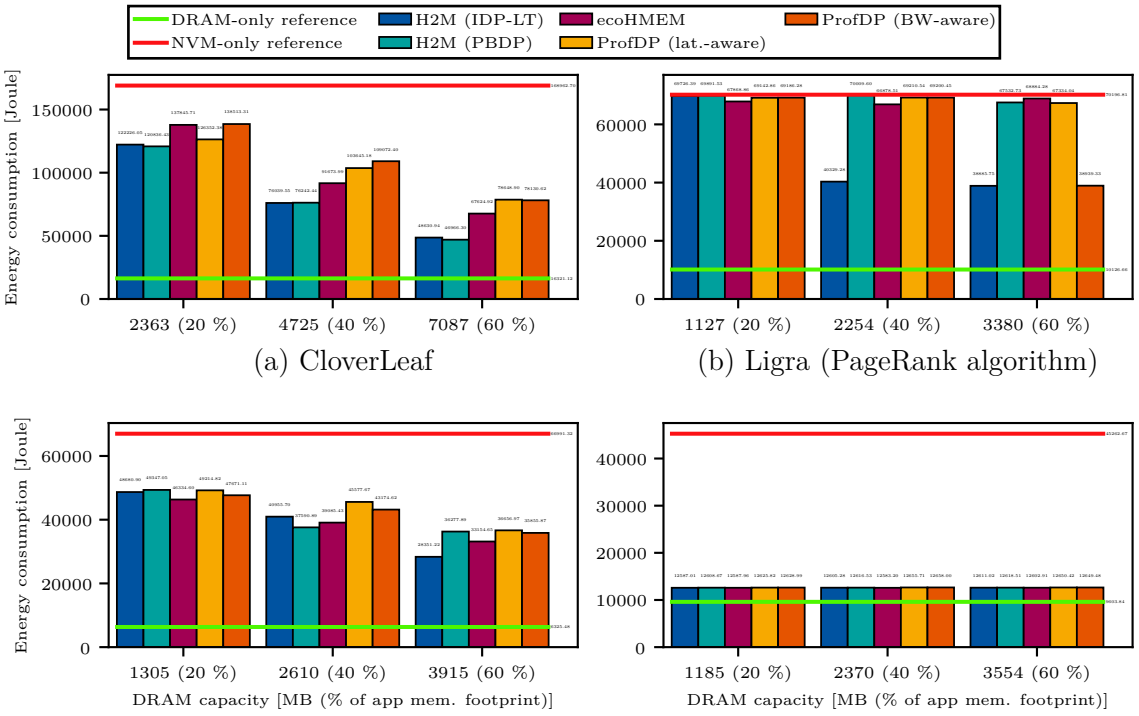
Figure C.2.: Power draw for the proxy applications (`DRAM` RAPL domain)



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

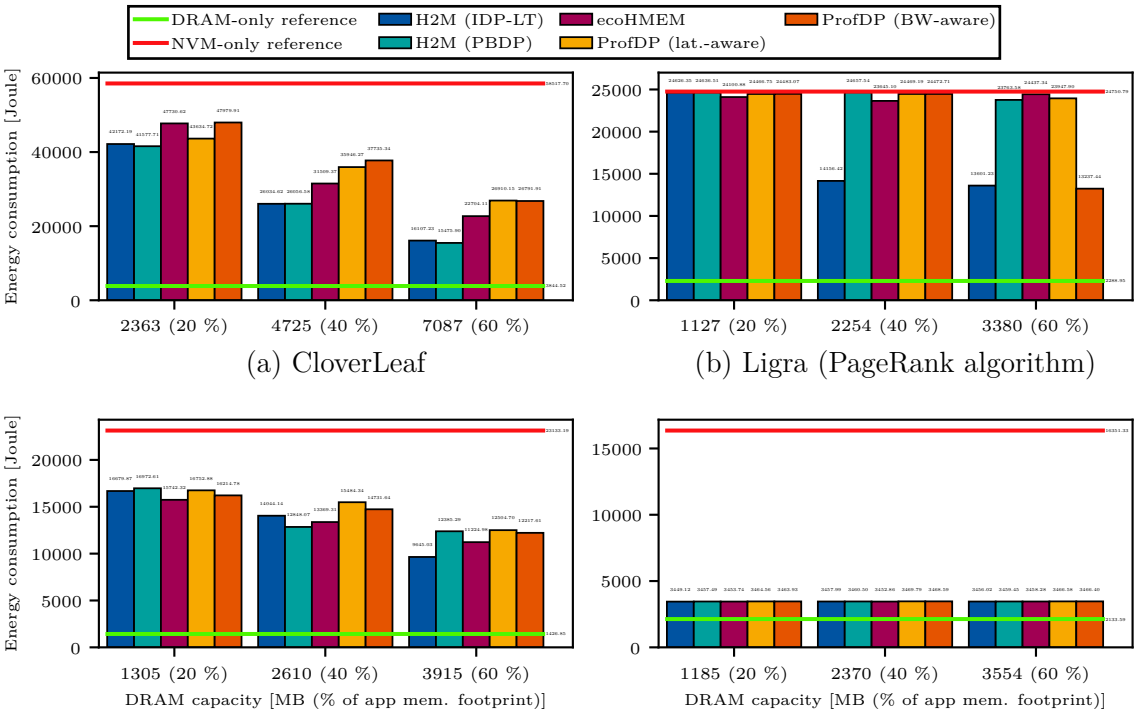
Figure C.3.: Power draw for the proxy applications (`Package` RAPL domain)



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

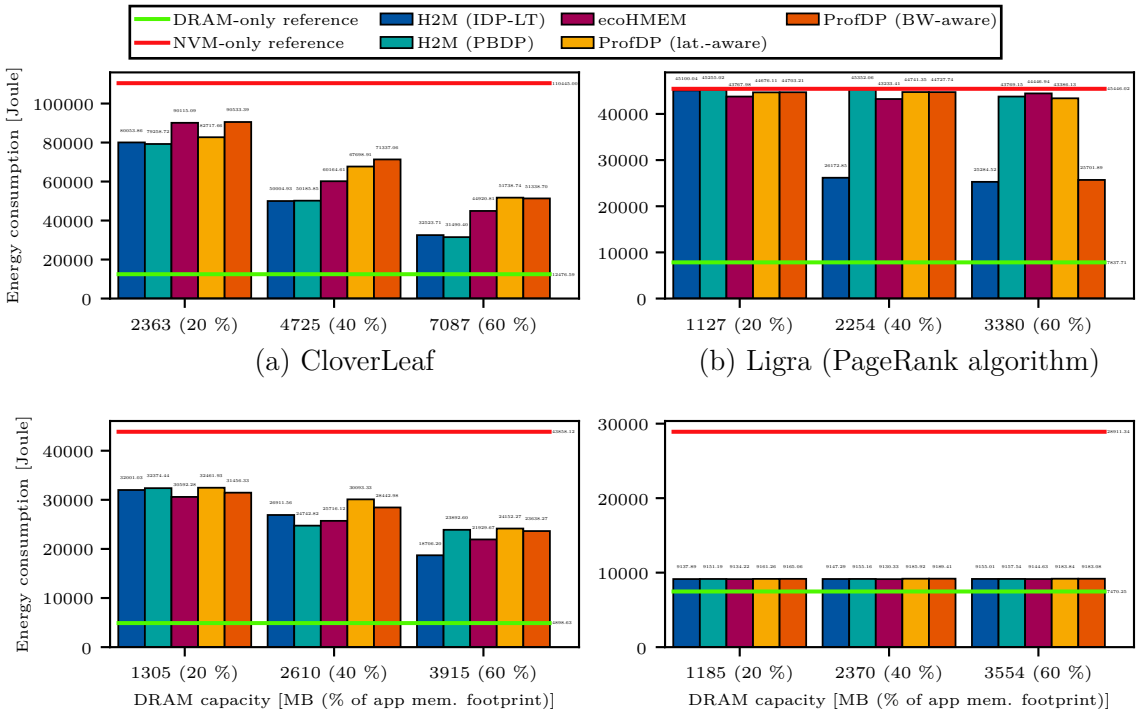
Figure C.4.: Energy consumption for the proxy applications (`Package` and `DRAM` RAPL domains combined)



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

Figure C.5.: Energy consumption for the proxy applications (`DRAM` RAPL domain)



(a) CloverLeaf (b) Ligra (PageRank algorithm)

(c) LULESH (d) XSBench

Figure C.6.: Energy consumption for the proxy applications (`Package` RAPL domain)

Bibliography

[1] Laksono Adhianto et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22.6 (Apr. 2010), pp. 685–701. ISSN: 1532-0626. DOI: 10.1002/cpe.1553.

[2] Alif Ahmed and Kevin Skadron. “Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation”. In: *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 167–172. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357574.

[3] Tyler Allen et al. “Performance and Energy Usage of Workloads on KNL and Haswell Architectures”. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Cham: Springer International Publishing, 2018, pp. 236–249. ISBN: 978-3-319-72971-8. DOI: 10.1007/978-3-319-72971-8_12.

[4] Lukas Alt et al. “An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory”. In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24)*. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 71–82. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645052.

[5] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. EECS-2006-183. Lawrence Berkeley National Laboratory, Dec. 2006. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf (visited on 09/26/2025).

[6] Francois Broquedis et al. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Feb. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67.

[7] Christopher Cantalupo et al. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*. Tech. rep. SAND2015-1862C. Sandia National Lab. Albuquerque, NM (United States), 2015. URL: https://www.osti.gov/servlets/purl/1245908 (visited on 09/26/2025).

[8] Barcelona Supercomputing Center. *Extrae*. (n.d.) URL: https://tools.bsc.es/extrae (visited on 05/29/2025).

[9] François Clautiaux, Boris Detienne, and Gaël Guillot. "An iterative dynamic programming approach for the temporal knapsack problem". In: *European Journal of Operational Research* 293.2 (Sept. 2021), pp. 442–456. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2020.12.036.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 87th ed. Mar. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 05/29/2025).

[11] Intel Corporation. *Intel® Memory Latency Checker (Intel® MLC)*. 2024. URL: https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html (visited on 09/09/2025).

[12] Intel Corporation. *Intel® Optane™ Persistent Memory (PMem)*. (n.d.) URL: https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html (visited on 05/23/2025).

[13] Intel Corporation. *Intel® VTune™ Profiler*. (n.d.) URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html (visited on 06/09/2025).

[14] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. (n.d.) URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (visited on 05/29/2025).

[15] Paul J. Drongowski. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Tech. rep. Nov. 2007. URL: https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/white-papers/AMD_IBS_paper_EN.pdf (visited on 09/25/2025).

[16] Subramanya R. Dulloor et al. "Data Tiering in Heterogeneous Memory Systems". In: *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 1–16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901344.

[17] Pouya Esmaili-Dokht et al. "A Mess of Memory System Benchmarking, Simulation and Application Profiling". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ISSN: 2379-3155. Nov. 2024, pp. 136–152. DOI: 10.1109/MICRO61859.2024.00020.

[18] Clément Foyer et al. "H2M: Towards Heuristics for Heterogeneous Memory". In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. ISSN: 2168-9253. Sept. 2022, pp. 498–499. DOI: 10.1109/CLUSTER51413.2022.00060.

[19] Manish Gupta et al. "Reliability-Aware Data Placement for Heterogeneous Memory Architecture". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 583–595. DOI: 10.1109/HPCA.2018.00056.

[20] Azzam Haidar et al. "Investigating power capping toward energy-efficient scientific applications". In: *Concurrency and Computation: Practice and Experience* 31.6 (2019), e4485. ISSN: 1532-0634. DOI: 10.1002/cpe.4485.

[21] Myeonggyun Han et al. "Hotness- and Lifetime-Aware Data Placement and Migration for High-Performance Deep Learning on Heterogeneous Memory Systems". In: *IEEE Transactions on Computers* 69.3 (Mar. 2020), pp. 377–391. ISSN: 1557-9956. DOI: 10.1109/TC.2019.2949408.

[22] Michael Allen Heroux, Jack Dongarra, and Piotr Luszczek. *HPCG Benchmark Technical Specification*. Tech. rep. SAND2013-8752. Sandia National Lab. Albuquerque, NM (United States), Oct. 2013. DOI: 10.2172/1113870.

[23] Intel Corporation. *FlexMalloc: Flexible Memory Allocation Tool*. (n.d.) URL: https://github.com/intel/flexmalloc (visited on 06/01/2025).

[24] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. Mar. 2019. URL: https://arxiv.org/abs/1903.05714v3 (visited on 05/22/2025).

[25] Tomislav Janjusic and Krishna Kavi. "Hardware and Application Profiling Tools". In: *Advances in Computers*. Vol. 92. Elsevier, 2014, pp. 105–160. DOI: 10.1016/B978-0-12-420232-0.00003-9.

[26] Hrvoje Jasak, Aleksandar Jemcov, and Željko Tuković. "OpenFOAM: A C++ Library for Complex Physics Simulations". In: *International Workshop on Coupled Methods in Numerical Dynamics (CMND)*. Dubrovnik, Croatia, Sept. 2007, pp. 47–66. URL: https://www.researchgate.net/publication/228879492_OpenFOAM_A_C_library_for_complex_physics_simulations (visited on 09/26/2025).

[27] Marc Jordà et al. "ecoHMEM: Improving Object Placement Methodology for Hybrid Memory Systems in HPC". In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2022, pp. 278–288. DOI: 10.1109/CLUSTER51413.2022.00040.

[28] Ian Karlin et al. "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 919–932. DOI: 10.1109/IPDPS.2013.115.

[29] Manolis Katsaragakis et al. "Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems". In: *ACM Transactions on Architecture and Code Optimization* 22.2 (July 2025), pp. 1–27. ISSN: 1544-3566. DOI: 10.1145/3736174.

[30] Dounia Khaldi and Barbara Chapman. "Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing". In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. Nov. 2016, pp. 12–20. DOI: 10.1109/LLVM-HPC.2016.007.

[31] Kashif N. Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3.2 (Mar. 2018), 9:1–9:26. ISSN: 2376-3639. DOI: 10.1145/3177754.

[32] Jannis Klinkenberg et al. "H2M: Exploiting Heterogeneous Shared Memory Architectures". In: *Future Generation Computer Systems* 148 (Nov. 2023), pp. 39–55. ISSN: 0167-739X. DOI: 10.1016/j.future.2023.05.019.

[33] Jannis Klinkenberg et al. "Phase-Based Data Placement Optimization in Heterogeneous Memory". In: *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2024, pp. 382–393. DOI: 10.1109/CLUSTER59578.2024.00040.

[34] Mohammad Laghari, Najeeb Ahmad, and Didem Unat. "Phase-Based Data Placement Scheme for Heterogeneous Memory Systems". In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Sept. 2018, pp. 189–196. DOI: 10.1109/CAHPC.2018.8645903.

[35] Robert Lasch et al. "Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory". In: *Proceedings of the VLDB Endowment* 15.11 (July 2022), pp. 2867–2880. ISSN: 2150-8097. DOI: 10.14778/3551793.3551837.

[36] Paul T. Lin et al. "Assessing a Miniapplication as a Performance Proxy for a Finite Element Method Engineering Applications Code". In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5374–5389. ISSN: 1532-0634. DOI: 10.1002/cpe.3587.

[37] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM SIGPLAN Notices* 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340. DOI: 10.1145/1064978.1065034.

[38] Andrew C. Mallinson et al. "CloverLeaf: Preparing Hydrodynamics Codes for Exascale". In: *A New Vintage of Computing: CUG2013*. Cray User Group, Inc., 2013. URL: https://www.cug.org/proceedings/cug2013_proceedings/includes/files/pap130.pdf (visited on 05/17/2025).

[39] John D. McCalpin. "Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors". In: *High Performance Computing*. Cham: Springer Nature Switzerland, 2023, pp. 403–413. ISBN: 978-3-031-40843-4. DOI: 10.1007/978-3-031-40843-4_30.

[40] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25. URL: https://www.researchgate.net/publication/51992086_Memory_bandwidth_and_machine_balance_in_high_performance_computers (visited on 06/09/2025).

[41] Aditya Narayan et al. "MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pp. 326–335. DOI: 10.1109/IPDPS.2018.00042.

[42] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[43] Scott Parker et al. "Early Evaluation of the Cray XC40 Xeon Phi System 'Theta' at Argonne". In: *Caffeinated Computing: CUG2017*. Cray User Group, Inc., 2017. URL: https://cug.org/proceedings/cug2017_proceedings/includes/files/pap113s2-file1.pdf (visited on 05/23/2025).

[44] Onkar Patil et al. "Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using Intel Optane DC Persistent Memory Modules". In: *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 288–303. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357541.

[45] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. "System Evaluation of the Intel Optane Byte-addressable NVM". In: *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 304–315. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357568.

[46] Ivy B. Peng et al. "Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2020, pp. 916–925. DOI: 10.1109/IPDPS47924.2020.00098.

[47] Ivy B. Peng et al. "Exploring the Performance Benefit of Hybrid Memory System on HPC Environments". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2017, pp. 683–692. DOI: 10.1109/IPDPSW.2017.115.

[48] Thomas Preisner et al. “Where are the Joules? Energy Demand Analysis of Heterogeneous Memory Technologies”. In: *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)*. New York, NY, USA: Association for Computing Machinery, Apr. 2025, pp. 60–66. ISBN: 979-8-4007-1470-2. DOI: 10.1145/3723851.3723857.

[49] Guillaume Raffin and Denis Trystram. “Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis”. In: *IEEE Transactions on Parallel and Distributed Systems* 36.1 (2025), pp. 96–107. DOI: 10.1109/TPDS.2024.3492336.

[50] Ganesan Ramalingam. “The Undecidability of Aliasing”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.5 (Sept. 1994), pp. 1467–1471. ISSN: 0164-0925. DOI: 10.1145/186025.186041.

[51] Solmaz Salehian and Yonghong Yan. “Evaluation of Knight Landing High Bandwidth Memory for HPC Workloads”. In: *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms (IA3 '17)*. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–4. ISBN: 978-1-4503-5136-2. DOI: 10.1145/3149704.3149766.

[52] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf (visited on 09/24/2025).

[53] Harald Servat et al. “Automating the Application Data Placement in Hybrid Memory Systems”. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2017, pp. 126–136. DOI: 10.1109/CLUSTER.2017.50. (Visited on 03/16/2025).

[54] Julian Shun and Guy E. Blelloch. “Ligra: A Lightweight Graph Processing Framework for Shared Memory”. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. Shenzhen, China: Association for Computing Machinery, Feb. 2013, pp. 135–146. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442530.

[55] Carl Staelin. “lmbench: an extensible micro-benchmark suite”. In: *Software: Practice and Experience* 35.11 (2005), pp. 1079–1105. DOI: 10.1002/spe.665.

[56] Aidan P. Thompson et al. “LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales”. In: *Computer Physics Communications* 271 (2022), e108171. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2021.108171.

[57] François Trahay et al. “NumaMMA: NUMA MeMory Analyzer”. In: *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. Eugene, OR, USA: Association for Computing Machinery, 2018, pp. 1–10. ISBN: 9781450365109. DOI: 10.1145/3225058.3225094.

[58] John R. Tramm et al. "XSBench - The development and verification of a performance abstraction for Monte Carlo reactor analysis". In: *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*. Kyoto, Japan, Sept. 2014. URL: https://www.researchgate.net/publication/354447461_XSBench_-_The_development_and_verification_of_a_performance_abstraction_for_Monte_Carlo_reactor_analysis (visited on 09/27/2025).

[59] Jan Treibig, Georg Hager, and Gerhard Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *39th International Conference on Parallel Processing Workshops*. San Diego, CA, USA, 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[60] Vasily Volkov. "Understanding latency hiding on GPUs". PhD thesis. Aug. 2016. URL: https://escholarship.org/content/qt1wb7f3h4/qt1wb7f3h4_noSplash_1e32f64125997ee6afa303a150338054.pdf (visited on 09/26/2025).

[61] Shasha Wen et al. "ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems". In: *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 263–273. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205320.

[62] Kai Wu, Jie Ren, and Dong Li. "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*. Nov. 2018, pp. 401–413. DOI: 10.1109/SC.2018.00034.

[63] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/system/files/fast20-yang.pdf (visited on 09/24/2025).

[64] Seongdae Yu, Seongbeom Park, and Woongki Baek. "Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems". In: *Proceedings of the 2017 International Conference on Supercomputing (ICS '17)*. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–10. ISBN: 978-1-4503-5020-4. DOI: 10.1145/3079079.3079092.

Index