Automating Context-Based Access Pattern Hint Injection for System Performance and Swap Storage Durability

SeongJae Park, Yunjae Lee, Moonsub Kim, and Heon Y. Yeom
Seoul National University

Abstract

Memory pressure is inevitable as the size of working sets is rapidly growing while the capacity of dynamic random-access memory (DRAM) is not. Meanwhile, storage devices have evolved so that their speed is comparable to the speed of DRAM while their capacity scales are comparable to that of hard disk drives (HDD). Thus, hierarchial memory systems configuring DRAM as the main memory and high-end storages as swap devices will be common.

Due to the unique characteristics of these modern storage devices, the swap target decision should be optimal. It is essential to know the exact data access patterns of workloads for such an optimal decision, although underlying systems cannot accurately estimate such complex and dynamic patterns. For this reason, memory systems allow programs to voluntarily hint their data access pattern. Nevertheless, it is exhausting for a human to manually figure out the patterns and embed optimal hints if the workloads are huge and complex.

This paper introduces a compiler extension that automatically optimizes a program to voluntarily hint its dynamic data access patterns to the underlying swap system using a static/dynamic analysis based profiling result. To our best knowledge, this is the first profile-guided optimization (PGO) for modern swap devices. Our empirical evaluation of the scheme using realistic workloads shows consistent improvement in performance and swap device lifetime up to 2.65 times and 2.98 times, respectively.

1 Introduction

As modern workloads such as clouds, big data, and machine learning are becoming more widespread, the size of working sets is rapidly growing [17, 29]. Compared to this, the size of the main memory (DRAM) in a single physical machine is even relatively shrinking [29]. Furthermore, cloud systems, one of the prevalent and promising computing environments, usually recommend memory overcommitment [11]. This trend implies that the main memory alone will not be able to accommodate all of the working set data. Fortunately, modern storage devices, such as solid state drives (SSD) or non-volatile memory (NVM), have rapidly evolved. These devices are fast enough to be compared even to DRAM and large enough to be even compared with HDD [2, 6]. Thus, computing systems utilizing hierarchical memory constructed with DRAM and fast storage devices will likely spread widely in the near future.

For this reason, various companies and researchers have proposed their design schemes [12, 14, 21] for such hierarchical memory systems. Despite the remarkable improvements that such designs provide, many of them cannot be instantly adapted in commodity systems because their new schemes usually require complex modifications. In contrast, the swap system can readily serve as a hierarchical memory as almost every commodity system has embedded and used it, even decades ago. That said, the swap system should be optimized because it is designed for HDDs rather than modern storage devices [24], and there are many differences in the characteristics of HDD and modern storage devices. Writes for those are slower than reads; they may even be worn out if the total number of writes for them exceeds a limit.

The most important part of the hierarchical memory system is the decision of a location for each data item. Items that will be frequently accessed should be in the main memory (DRAM), and the number of writes to the auxiliary memory (swap device) should be minimized. The swap system in the Linux kernel has been employing a pseudo least recently used (LRU) technique [18] for swap target decision, and various alternative schemes also exist [13, 26, 27]. Nevertheless, such estimation-based schemes cannot make the optimal decision for programs with unpredictable dynamic data access patterns.

For this reason, most operating systems provide special system calls [3, 5] that allow user programs to voluntarily hint their data access patterns for a specific memory region to the underlying memory system. For example, the `mlock()` system call [5] forces page frames for a specified memory region to be locked in the main memory. The correct use of these system calls can significantly improve performance

of memory-intensive workloads and the durability of swap devices. However, it is exhaustive for a human to manually analyze detailed dynamic data access patterns of a huge and complex program. Worse yet, modifying the program to voluntarily provide the optimal hints to an underlying swap system with minimal overhead is excessively hard as well.

We introduce a data access pattern hint injecting compiler extension, DAPHICX, which aims to automate such tasks. It receives a program source code and analyzes its data access patterns via static/dynamic analysis based profiling. In detail, the profiling is based on program execution contexts because the dynamic data access pattern of a code section depends on its execution context. Though the context-based profiling guarantees high accuracy, the profiling result is too large and verbose. Therefore, directly providing the information to the underlying swap system can result in an excessive overhead. To mitigate the overhead, DAPHICX optimizes the large amount verbose information into compact and meaningful hints. Using the hints, it injects system call invocations into the program so that the generated program voluntarily notifies the optimized hints to the underlying swap system. To our best knowledge, this is the first profile-guided optimization (PGO) [9] for a swap system employing modern storage devices.

We implemented the prototype based on LLVM [8] and evaluated it with a number of realistic memory-intensive workloads. The evaluation results showed consistent improvement of performance and swap device lifetime of up to 2.65 and 2.98 times, respectively.

2 DAPHICX: Data Access Pattern Hint Injecting Compiler eXtension

The overall architecture of a system employing the DAPHICX is shown in Figure 1. Because the DAPHICX is a compiler extension, it essentially works as a part of a compiler, which receives the source code of a program as an input and returns an executable binary file as an output. The DAPHICX profiles dynamic the data access patterns of a given program, decides which hints to give to the underlying swap system, and injects system call invocation code into the executable binary that transfers the hints to the underlying swap system. After, the hint is transferred to the underlying swap system when the hint-injected program is executed. Then, the swap system places data in the DRAM or swap devices based on the received hints.

2.1 Data Access Pattern Profiling

A program is composed with multiple code sections (e.g., functions or loops), and each section has unique characteristics. Further, even a single code section can have multiple characteristics depending on its execution context. Because the data access pattern also depends on the context, the DAPHICX first extracts the execution context information from the received program via static analysis. In detail, it constructs an execution flow tree by following the execution flow of a given source code. We call the tree a *context tree*. Listing 1 and Figure 2 show an example program source code and its context tree, respectively. In this example, executions of `bar()` from `foo()` and from `main()` are distinguished as different contexts.

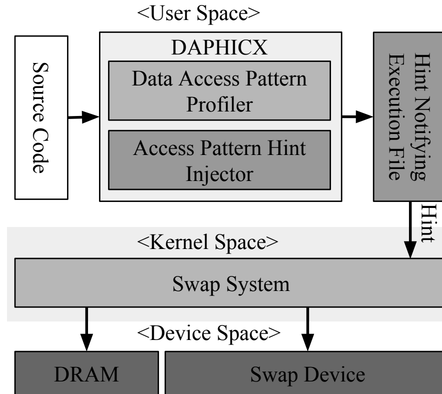

Figure 1: A system employing the DAPHICX.

After the extraction of the context tree, it identifies memory objects in the program based on the contexts where they are allocated. In other words, a memory region allocated from a different execution context is identified as a different memory object. Finally, it injects memory access profiling code, which records the number of accesses to each memory object during the execution of each context into the program and then executes the profiling code injected program. After this execution completes, users can finally access the detailed data access patterns of the program.

2.2 Small Contexts to Meaningful Phases

One straightforward approach is notifying the data access pattern of each context to the underlying swap system just before the context starts execution. However, because the hinting operation itself also has an inherent overhead, this naive approach can induce significant overhead if the execution time of each context is not long enough to conceal the overhead of the hinting operation. In actuality, such cases are common. For example, the workload of the simulation environment for quantum chromodynamics [1] that we use in Section 3 completes running in about 300 seconds, and it changes its context more than a billion times. This measurement implies the fact that each context in the application consumes less than one microsecond on average. Thus, if notifying a data access pattern for a context requires more than one microsecond, execution of the hint-injected binary will be dominated by the execution of those hinting opera-

```
1 int bar() {
2     return 42;
3 }
4
5 void foo() {
6     for (int i = 0; i < 3; i++)
7         bar();
8 }
9
10 int main(void) {
11    foo();
12    return bar();
13 }
```

Listing 1: An example code.

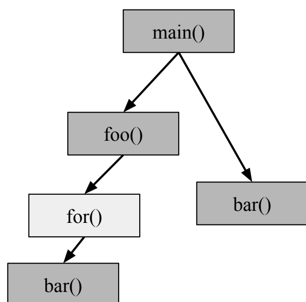

Figure 2: A context tree of the example code.

tions. To mitigate such overheads, the DAPHICX integrates small or meaningless contexts into bigger and meaningful ones, which we call phases, based on two metrics. The first metric is the execution time of each context. If the execution time of a context is shorter than a pre-defined threshold, the context is merged into its parent context until the execution time of the merged context exceeds the threshold. The second metric is the data access pattern similarity. If a context and its parent context have similar data access patterns, we merge them into one phase to further minimize meaningless hint notification overheads. Finally, the DAPHICX injects the data access pattern hinting code for each phase into the program.

2.3 Hinting Data Access Pattern

Among system calls, we use `mlock()` and `munlock()` system calls [5] to hint the data access pattern to the swap system. In detail, those system calls are used to notify important and unimportant memory objects respectively, as `mlock()` guarantees page frames for the specified memory region to be locked in the main memory while `munlock()` invalidates those guarantees.

For the selection of memory objects to be locked in, we calculate a priority for each memory object based on the following two observations. By locking an object, the program (1) takes benefits from the increased DRAM hit by the number of accesses to the object and (2) sacrifices available memory by the size of the object. Consequently, rarely accessed and/or large memory objects should not be locked in and vice versa. Based on this simple idea, we designed Equation 1 for the prioritization of memory objects. The priority of an object ($priority(object)$) becomes greater as the number of accesses to the object ($object.nr_accesses$) increases and the size of the object ($object.mem_size$) becomes smaller. α and β control the growth rate of each metric's impact, respectively.

$$priority(object) = \frac{object.nr_accesses^{\alpha}}{object.mem_size^{\beta}} \quad (1)$$

After assigning a priority score to every object, we select a group of important memory objects to be placed in the main memory for each phase. The total size of the selected objects should be equal to or smaller than that of the available main memory, and the sum of the priority of those objects should be the highest among every possible selection. This requirement is similar to the knap-sack problem [22]. Because the knapsack problem is NP-hard, and we need to trade-off between the overhead and accuracy, we designed another straightforward algorithm: the greedy knapsack (Algorithm 1).

Algorithm 1: The greedy knapsack for the objects' selection.

Input: $objects[], mem_limit$
1. $to_lock \leftarrow [\,]$
2. $size_locked \leftarrow 0$
3. Sort $objects$ by $priority$
4. **for each** Object o in $objects$ **do**
5.   **if** $size_locked + o.mem_size < mem_limit$ **then**
6. $objects$.remove(o)
7. $size_locked \leftarrow size_locked + o.mem_size$
8. to_lock.append(o)
9.   **else if** $size_locked < mem_limit \times 0.9$ **then**
10.   Split o in half
11.   go to 3
12.   **end**
13. **end**
14. **return** to_lock

For each phase, it receives (1) a list of objects ($objects$) and (2) the available memory size (mem_limit) as input and returns a set of objects to be locked into (to_lock) the available memory during the execution of the phase. Lines 1–2 initialize variables to be used. to_lock is a list of objects to be locked in, and $size_locked$ is the total size of the objects selected to be locked in. Line 3 sorts the objects by their priority score in descending order. In the for loop spanning lines 4–12, if the size of the object that has the highest priority in the unselected list ($objects$) is small enough to be located in the main memory with previously selected objects (line 5), the object is moved from the unselected objects list to the selected objects list (lines 6–8).

If the condition in line 5 is false (the highest priority object is too large to be placed in the available memory with the previously selected objects), and the memory utilization rate with the currently selected objects is lower than 90% (line 9), it splits the current object in half (line 10) and goes back to the priority-based sorting task (line 11) and restarts the loop. As a result, about 90% of the main memory is filled with the highest priority memory objects.

After this selection, the DAPHICX finally injects the hinting code into the target program. The injected code is executed just before the start of each phase of the program. It calls `munlock()` for objects not selected to be locked in for the phase if necessary and locks objects selected to be locked in via `mlock()`. Then, the injected hint code is completed and the original application code for the phase resumes execution.

3 Evaluation

We implemented a prototype of the DAPHICX based on LLVM [8] for empirical evaluations. The implementation consists of about 5,000 lines of code for profiling, 1,300 lines of code for hint injection, and 200 lines of code for hinting object selection.

3.1 Evaluation Setup

The server we use for evaluation runs the Linux kernel v4.14 and equips an Intel Xeon E7-8837 processor, 128 GB DRAM, and an Intel Optane SSD as a swap device. We choose eight benchmarks from the SPEC CPU 2006 benchmark suite [7]. In the selected benchmarks, both memory-intensive and compute-intensive workloads are mixed [20].

We simulate memory pressure by reducing the size of the available memory under the working set size of each workload using the memory resource controller [4] of the `cgroups` in the Linux kernel. We increase the memory shortage to up to 30% of the working set size because Openstack, one of the widely adopted cloud systems, officially recommends a 1.5:1 memory overcommitment [11]. This would not be a common case, though, because Openstack also attempts to minimize these situations. We heuristically set thresholds and constants, such as the *time threshold* and *data access pattern similarity threshold* for the context merging and α and β of Equation 1. The available memory size for Algorithm 1 is statically fixed as only 70% of each workload's working set size. This pessimistic setup is reasonable for cloud administrators who need to prepare for unpredictable memory pressures. We repeatedly run every workload three times and use the average result to minimize the measurement error.

3.2 Evaluation Result

Figures 3(a) and 3(b) show that the DAPHICX achieved improvements in the performance and in the lifetime of the swap device for each workload, respectively. We define the performance improvement as the runtime ratio of the original version to the hint-injected version (runtime speedup) and the lifetime improvement as the ratio of the number of swap out events (number of reduced writes).

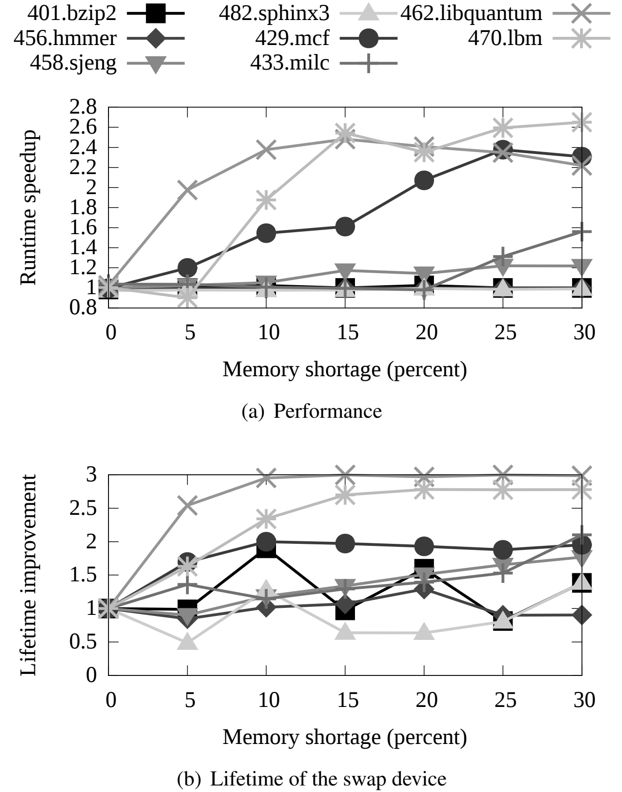

Figure 3: Hint-injected version achieved improvements in the performance and in the lifetime of the swap device for each workload with varying amount of memory pressure.

The DAPHICX consistently improves the performance of every workload. The amount of improvement grows as the memory shortage becomes severe. It shows a significant performance improvement (over 40% and up to 265%) for five workloads (482.sphinx3, 429.mcf, 433.milc, 462.libquantum, and 460.lbm) while the other workloads (401.bzip2, 456.hmmer, and 458.sjeng) shows neither improvement nor degradation. The DAPHICX shows no performance degradation for any case, including the absence of memory pressure due to our optimizations and tradeoffs. Nevertheless, 470.lbm shows a slight performance drop under a memory shortage of 5% though the amount is negligible.

Our work also consistently reduces the number of writes to the swap device at up to 2.98 times. Interestingly, 482.sphinx3 shows a swap device lifetime decrease when small memory pressures are induced though it eventually improved the lifetime about 50% under 30% memory pressure. The workload has plethora of memory objects, so the data for the hint itself

becomes even larger than the working set size (about 50 MB) of the original workload. As a result, the working set size of the hint-injected version becomes significantly larger than that of the original version, and more swap out events occur. That said, the hint-injected version still shows no performance degradation for any case and eventually improves swap device lifetime with 30% memory pressure because the hint code itself is effectively optimized.

4 Related Works

A couple of context-based write grouping schemes [19, 23] for SSD have been proposed. Those schemes utilize context information, which is conceptually similar to ours, for the classification of data items or write requests with similar update patterns. These approaches, though, capture the context information via dynamic stack tracing while we extract it with a static program analysis.

A number of novel schemes for data placement automation in heterogeneous memory systems [16, 28, 30] exist. Such works profile data access patterns, calculate priority, and discern the optimal place for each or classes of memory objects. Though their target memory structure, which is heterogeneous, is different than ours, which is hierarchical, the basic ideas behind theirs and ours are somewhat similar and compatible. Nevertheless, their works are not aware of the dynamic data access pattern for each phase of a given workload. Thus, the dynamic data access pattern awareness of DAPHICX is its key difference.

Lagar-Cavilla *et al.* [15,25] introduced how Google utilizes their hierarchical memory system for their cluster environments. The system equips an in-memory compressed block device [10] as a swap device and proactively swaps out idle pages to minimize memory pressure. To classify the idle pages, it tracks page table access bits with a dedicated CPU core and tunes logic parameters via machine learning. The access pattern tracking overhead can be arbitrarily high as the working set size of given workloads grows, and its time granularity of proactive reclamation, two minutes, is somewhat too coarse. Meanwhile, our approach incurs almost no production runtime overhead and provides finely grained hints with only negligible runtime overhead.

5 Future works

In spite of the demonstrated improvement of DAPHICX, a few things still remain for future work. The object priority calculation algorithm of the DAPHICX has the naive assumption that the memory region inside each memory object is uniformly accessed. Though such access patterns are common, some programs optimized for special purposes can access sub-memory regions in a single memory object with different patterns. In such a case, the prioritization algorithm of DAPHICX can generate inappropriate hints. We intend to profile the access pattern inside an object and apply the result in the future.

Offsets for merging contexts are important for the effective trade-off between overhead and accuracy. However, the offsets in this paper are only selected by a heuristic. If the offsets are too small, the program will have an overmuch phase and a high overhead for hint notification and vice versa. Therefore, we will develop an algorithm or a model for selecting the optimal offset automatically.

Though we focused on the swap system due to its unique stability and wide avalability, other multi-tier memory systems will eventually be matured. Expanding our schme to such other systems would only require affordable efforts because of its generality. In particular, the priority calcuation model is adoptable for a general multi-tier memory and is tunable for a specific one. All the things required to the systems for adoption of our scheme are `mlock()`-like user level access pattern hint primitives and general programming model that our static analysis could be applied.

6 Conclusion

Data-intensive workloads with huge working sets are becoming widespread, and high-end storage devices today are even able to be compared with the speed of DRAM and the capacity of HDD. These recent trends intimate the widespread use of hierarchical memory in the near future. In particular, the swap system will be widely adopted for its availability and ease of use. Nevertheless, the modern storages are still obviously slower than the DRAM, and those storages could be worn out if the number of writes exceeds a limit. Thus, the swap target decision should be as optimal as possible. For this reason, memory systems allow programmers to voluntarily hint data access patterns of the program though analyzing the patterns and injecting hints into the program is exhaustively difficult for a human.

To automate the exhaustive tasks, we introduced a data access pattern hint injecting compiler extension, DAPHICX, which applies profile-guided optimization for accurate and efficient swap target decisions. It profiles a program to get the data access pattern of each program context, transforms the detailed but verbose information to efficient and meaningful hints, and injects a code for the hints into the program. Our evaluation of a prototype achieved up to 2.65 times speedup and up to 2.98 times swap device lifetime improvement.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015M3C4A7065646 and NRF-2016M3C4A7952587).

7 Discussions

Because this paper introduces the early results of ongoing research rather than a completed study, we are looking for feedback and opinions about the following topics for the future development of this work.

Adoption of the swap system into modern computing area. Because the swap system was traditionally considered harmful, many environments recommend or are even forced to disable the swap system. Kubernetes is a good example. Though we believe that the ease of use of the swap system, the trend of data exploitation, and the evolvement of hardware (DRAM and modern storage devices) implies widespread use of the swap system in the near future, others could have different opinions. Therefore, we would like to receive feedback about the prospect and the opinions of others and/or their usage experience concerning the swap system, from both academic and industry experts.

Feasibility of our memory object priority calculation and heuristics. The equation for memory object priority, constants in the equation that leverage weights of the metrics, and the offsets for tradeoffs between accuracy and overhead are the most important keys of our scheme. Though the equation and values we used in this paper have been carefully developed and successfully adapted to multiple realistic workloads, these are based on straightforward insights and leave the decision of the optimal value for important constants and offsets to users. Though we have already started to develop more sophisticated schemes, we want to hear the others' feedback and recommendations concerning known models.

Additional requirements for adoption into other areas. We manifestly specified the target environments that our scheme is aiming to be adopted within. That said, we believe that this whole of or part of the technique could be useful for other environments we did not consider. For example, mobile, IoT, real-time, and firmware levels could be such targets. Because there could be many storage experts from various areas, we would like to request comments and feedback about the challenges and requirements of our techniques for extended adoption into other areas.

References

[1] 433.milc, SPEC CPU2006 Benchmark Description. https://www.spec.org/cpu2006/Docs/433.milc.html.

[2] Intel's new Optane SSDs are superfast and can even work as extra RAM. https://www.theverge.com/circuitbreaker/2017/10/31/16582018/intel-optane-p900-ssd-fast-dram-nand-flash-memory-desktop-computer.

[3] madvise(2) - Linux manual page. http://man7.org/linux/man-pages/man2/madvise.2.html.

[4] Memory Resource Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[5] mlock(2) - Linux manual page. http://man7.org/linux/man-pages/man2/mlock.2.html.

[6] Samsung unveils world's largest SSD with whopping 30TB of storage. https://www.theverge.com/circuitbreaker/2018/2/20/17031256/worlds-largest-ssd-drive-samsung-30-terabyte-pm1643.

[7] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[8] The LLVM Compiler Infrastructure. https://llvm.org/.

[9] Using Profile-Guided Optimization (PGO). https://source.android.com/devices/tech/perf/pgo.

[10] zswap.txt. https://www.kernel.org/doc/Documentation/vm/zswap.txt.

[11] Overcommitting CPU and RAM. https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html, 2018.

[12] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News*, 45(1):631–644, 2017.

[13] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[14] Katherine Bourzac. Has intel created a universal memory technology?[news]. *IEEE Spectrum*, 54(5):9–10, 2017.

[15] Jonathan Corbet. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/, 2019.

[16] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

[17] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds. In *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, volume 47, page 37, New York, New York, USA, 2012. ACM Press.

[18] Mel Gorman. *Page Frame Reclamation*. Prentice Hall Upper Saddle River, 2004. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[19] Keonsoo Ha and Jihong Kim. A program context-aware data separation technique for reducing garbage collection overhead in nand flash memory. *Proc. 7th IEEE SNAPI*, 2011.

[20] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2007.

[21] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pvm: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 13. ACM, 2016.

[22] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to np-completeness of knapsack problems. In *Knapsack problems*, pages 483–493. Springer, 2004.

[23] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed ssds using program contexts. In *17th USENIX Conference on File and Storage Technologies FAST 19*, pages 295–308, 2019.

[24] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A new linux swap system for flash memory storage devices. In *2008 International Conference on Computational Sciences and Its Applications*, pages 151–156. IEEE, 2008.

[25] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 317–330, New York, NY, USA, 2019. ACM.

[26] Zhan-sheng Li, Da-wei Liu, and Hui-juan Bi. Crfp: a novel adaptive replacement policy combined the lru and lfu policies. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 72–79. IEEE, 2008.

[27] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[28] Gaku Nakagawa and Shuichi Oikawa. Data placement based on data semantics for nvdimm/dram hybrid memory architecture. *CLOUD COMPUTING 2017*, page 109, 2017.

[29] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, page 16. ACM, 2018.

[30] Harald Servat, Antonio J Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136. IEEE, 2017.