M.S. THESIS

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

FEBRUARY 2018

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

M.S. THESIS

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

FEBRUARY 2018

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

지도교수 염헌영

이 논문을 공학석사 학위논문으로 제출함

2017 년 11 월

서울대학교 대학원

컴퓨터 공학부

김화정

김화정의 공학석사 학위논문을 인준함

2018 년 01 월

위 원 장	엄현상	(인)
부위원장	염헌영	(인)
위 원	전병곤	(인)

Abstract

Phase change memory (PCM) is one of the promising non-volatile memory (NVM) technologies since it provides both high capacity and low idle power consumption. However, relatively slow access latency is one of the major challenges in using PCM as main memory. Therefore, in recent researches, it is attempting to construct heterogeneous memory systems by combining such NVM with DRAM. One of the major problems with using those systems is placing the data in the appropriate type of memory. In this paper, we propose an object placement method to address data placement problem in heterogeneous memory systems. With context-aware object profile information, we could dynamically detect memory access patterns of objects and determine the proper memory to place the objects on. We demonstrate the effectiveness of the proposed method by simulating memory access latency and energy consumption using the four selected workloads of the SPEC benchmark.

Keywords: Object Placement, Heterogeneous Memory System, Context-Aware Object Profiling

Student Number: 2016-21198

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Phase change memory (PCM) is one of the emerging non-volatile memory (NVM) devices. PCM has a longer access latency than DRAM and consumes much energy to read and write data [1, 2, 3]. Despite its limitations in latency and bandwidth, it still offers appealing attributes. Its density is much higher than DRAM and provides near-zero idle power consumption. Thus, several studies have been done to construct heterogeneous memory systems to overcome performance overhead and take advantage of NVM [4, 5].

S.R. Dulloor et al. [4] proposed data classification and tiering techniques under hybrid memory architecture. Researchers in [4] used an offline profiling tool (PIN) to understand access patterns of different data structures of applications. On the other hand, Kai Wu et al. [5] introduced Unimem to dynamically place objects on heterogeneous memory systems. Unimem focuses on memory access patterns of applications that operate in iterative structures. It manages data placement based on runtime profiling and performance models and targets high-performance computing (HPC) systems.

In this paper, we propose an object placement modeling method using the knowledge of access patterns of objects to the memory. To capture memory access patterns of objects allocated by an application, we use CAMP [6] framework. It is a framework that provides context-aware memory dependency information, and we will provide a more detailed description in Section 2.2. Using CAMP framework, we could collect memory access information of each object and allocate to appropriate memory at the compiler-level. We propose a modeling method that demonstrates the effect of placing objects on various memory systems in terms of latency and energy consumption, depending on the memory access patterns of the objects allocated by the application. To verify the effect of object placement, several simulations were performed assuming different memory systems. Simulation results show that appropriate objects placement in a heterogeneous memory system could achieve a DRAM-like latency while minimizing the overall energy consumption of the system.

Chapter 2

Background and Motivation

This chapter briefly describes the structure of the heterogeneous memory system used in this paper. Also, we introduce the profiling tool and methods that are used to capture the behavior of objects allocated to applications during runtime.

2.1 Heterogeneous Memory Systems

Different to DRAM, PCM has asymmetric read and write characteristics, as has been shown in previous studies [1, 2, 3]. Table 2.1 compares latency and energy consumption for DRAM and PCM access [3, 7]. Using PCM instead of DRAM requires up to 50 times the latency and 6 times the energy for a single write operation. However, PCM has some attractive characteristics in spite of such performance limitations. First of all, PCM maintains data persistently even in the event of a power failure. PCM also has a higher density than DRAM and has near-zero idle power consumption. Therefore, several studies have been conducted to compose heterogeneous memory systems utilizing PCM as well as

a DRAM [8, 9].

	DRAM	PCM
Page Size	64 B	64 B
Read Latency	20-50 ns	$\sim$ 50 ns
Write Latency	20-50 ns	$\sim$ 1 μs
Read Energy	0.8 J/GB	1 J/GB
Write Energy	1.2 J/GB	6 J/GB
Idle Power	$\sim$ 100 mW/GB	$\sim$ 1 mW/GB

Table 2.1: Latency and Energy Consumption Comparison between DRAM and PCM access

2.2 Context-Aware Memory Profiling

The CAMP framework is a Context-Aware Memory Profiling framework that traces program memory dependencies using full context information of a program, such as a call site stack and loop nesting levels. The CAMP statically detects all possible contexts in the program and assigns a unique ID to each context. It also generates a static context tree for the program, as shown in the Figure 2.1. Figure 2.1b shows the context tree for the example program shown in Figure 2.1a.

2.3 Object Profiling and Placement

We use the CAMP framework [6] to understand memory access patterns of applications. The CAMP framework is implemented on top of the LLVM compiler infrastructure [10]. With the LLVM infrastructure, it is easy to apply compiler-level optimization features. The original CAMP framework provides profiling results for all possible load and store dependencies of objects allocated by the

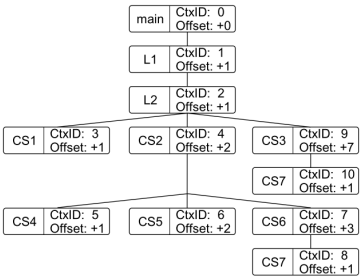
application. We added a few lines of code to the CAMP framework to identify all objects access patterns in the application. We also implemented additional optimization pass in the CAMP framework to dynamically place objects on the heterogeneous memory system at the compiler-level.

We named the additional pass as ctx-obj-placement. The pass automatically allocates objects to the heterogeneous memory system, which are transparent to the application. In the pass, it first reads the object access pattern profiling results generated by the previous dynamic profiling execution. The pass then calculates the amount of write access to each object and determines the threshold at which to place the object on the heterogeneous memory system. During the runtime of the application, the pass automatically catches memory allocation requests from the application and allocates space in the appropriate memory.

Figure 2.2 summarizes the application profiling process of the CAMP framework.

```
1  int getValue(Node *n) {
2    return n->value;                  // LD1
3  }
4
5  void setValue(Node *n, int v) {
6    if(isValid(v))                    // CS7
7      n->value = v;                   // ST1
8  }
9
10 int work(Node *n) {
11   int v1 = getValue(n);             // CS4
12   int v2 = update(v1);              // CS5
13   setValue(n, v2);                  // CS6
14   return v2;
15 }
16
17 void main() {
18   for(int t = 0; t < T; t++) {      // L1
19     for(int i = 0; i < N; i++) {    // L2
20       int s = getValue(sum[t]);     // CS1
21       int v = work(nodes[i]);       // CS2
22       s = s + v;                    // ADD
23       setValue(sum[t], s);          // CS3
24     }
25   }
26 }
```

(a) Example Program



(b) Context Tree for the Example Program

Figure 2.1: Context-Aware Memory Dependence Profiling Example

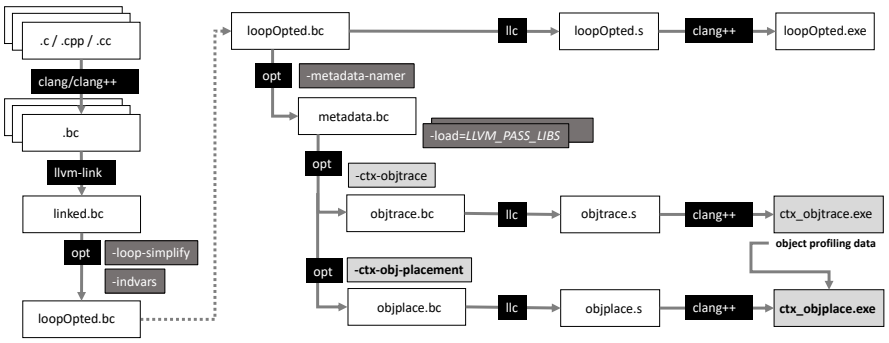


Figure 2.2: Context-Aware Object Trace using LLVM

Chapter 3

Object Placement Modeling

In this chapter, we explain the modeling method to simulate the performance of the application in terms of latency and energy consumption on the heterogeneous memory system. Depending on the modeling method, a detailed description of how to determine the appropriate memory to allocate for each object in the application is provided at the end of this chapter.

3.1 Basic Assumptions

To simplify the object placement problem, we establish following assumptions.

Assumption 1. *Caching effects on objects are the same in various memory systems.*

Assumption 2. *The difference in access latency between DRAM and PCM is maximum. That is, we set the DRAM read and write to 20 ns and the PCM to 50 ns and 1 μs.*

Assumption 3. *The application latency is proportional to the memory access latency of objects referenced during runtime. Thus, the performance comparison between different types of memory is based on the memory access pattern of all objects allocated by the application.*

3.2 Latency Modeling

From the dynamic profiling results of each object, we can figure out how many times each object is loaded and stored while the application is running. With the size of the allocated object, latency prediction modeling is as follows. The Equation 3.1 and the Equation 3.2 represent the load and store latency for each object allocated to a heterogeneous memory system.

The load latency of an object ($LatLD_{obj}$) is determined by the read latency of DRAM ($LatLD_{DRAM}$), the number of load instructions for DRAM of object ($NumLD_{DRAM}$), the read latency of PCM ($LatLD_{PCM}$), and the number of load instructions for PCM of object ($NumLD_{PCM}$). Likewise, the store latency of an object ($LatST_{obj}$) is determined by the write latency of DRAM ($LatST_{DRAM}$), the number of store instructions for DRAM of object ($NumST_{DRAM}$), the write latency of PCM ($LatST_{PCM}$), and the number of store instructions for PCM of object ($NumST_{PCM}$).

$$LatLD_{obj} = LatLD_{DRAM} \times NumLD_{DRAM} + LatLD_{PCM} \times NumLD_{PCM} \tag{3.1}$$

$$LatST_{obj} = LatST_{DRAM} \times NumST_{DRAM} + LatST_{PCM} \times NumST_{PCM} \tag{3.2}$$

The Equation 3.3 represents the latency of an application. It is the sum of load and store latency of each object allocated while the application is running.

$$Latency_{application} = \sum_{obj}(LatLD_{obj} + LatST_{obj}) \tag{3.3}$$

3.3 Energy Consumption Modeling

Energy consumption modeling considers the size of each object allocated to the heterogeneous memory system, as opposed to the latency prediction modeling introduced in Section 3.2. The Equation 3.4 and the Equation 3.5 represent the load and store energy consumption of each object allocated to the heterogeneous memory system.

The load energy consumption of an object ($EnergyLD_{obj}$) is determined by the load energy consumption of DRAM ($EnergyLD_{DRAM}$), the number of load instructions for DRAM of the object ($NumLD_{DRAM}$), the size of the object allocated on DRAM ($Size_{DRAM}$), the load energy consumption of PCM ($EnergyLD_{PCM}$), the number of load instructions for PCM of the object ($NumLD_{PCM}$), and the size of the object allocated on PCM ($Size_{PCM}$). Likewise, the store energy consumption of an object ($EnergyST_{obj}$) is determined by the store energy consumption of DRAM ($EnergyST_{DRAM}$), the number of store instructions for DRAM of the object ($NumST_{DRAM}$), the size of the object allocated on DRAM ($Size_{DRAM}$), the store energy consumption of PCM ($EnergyST_{PCM}$), the number of store instructions for PCM of the object ($NumST_{PCM}$), and the size of the object allocated on PCM ($Size_{PCM}$).

$$\begin{aligned} EnergyLD_{obj} = & EnergyLD_{DRAM} \times NumLD_{DRAM} \times Size_{DRAM} \\ & + EnergyLD_{PCM} \times NumLD_{PCM} \times Size_{PCM} \end{aligned} \tag{3.4}$$

$$\begin{aligned} EnergyST_{obj} = & EnergyST_{DRAM} \times NumST_{DRAM} \times Size_{DRAM} \\ & + EnergyST_{PCM} \times NumST_{PCM} \times Size_{PCM} \end{aligned} \tag{3.5}$$

The Equation 3.6 represents the energy consumption of an application. It is the sum of $EnergyLD_{obj}$ and $EnergyST_{obj}$ for all objects allocated while the application is running.

$$Energy_{application} = \sum_{obj} (EnergyLD_{obj} + EnergyST_{obj}) \tag{3.6}$$

3.4 Idle Power Consumption Modeling

We also simulated the idle power consumption of the memory system while an application is running, based on the simulated latency of the application and the total object size allocated during application runtime. The Equation 3.7 represents the idle power consumption of the memory system.

In the equation, $Latency_{application}$ is calculated from the Equation 3.3. $Size_{DRAM}$ and $Size_{PCM}$ are the total object size allocated to DRAM and PCM while the application is running.

$$\begin{aligned} IdlePower_{memory} = & Latency_{application} \times IdlePower_{DRAM} \times Size_{DRAM} \\ & + Latency_{application} \times IdlePower_{PCM} \times Size_{PCM} \end{aligned} \tag{3.7}$$

3.5 Object Placement Decision

As mentioned in previous chapters, the performance difference between DRAM and PCM depends primarily on the total amount of write instructions of the application. Therefore, we determine object placement based on the number and

the size of write instructions for each object. For all objects allocated by the application, we first calculate the total amount of write access to the memory of each object. We then choose proper value as the threshold.

The compiler pass first scans through all objects and compare the total amount of write access of each object to the threshold value during initialization of the application. The object is placed in DRAM when the total amount is greater than the threshold, otherwise, it is placed in PCM. Table 4.2 presents the result of object placement based on profiling information with the median value as the threshold.

Thresholds can vary depending on the characteristics of the application. In the paper, we simulated with two different threshold values, zero and the median. All simulation results are given in Chapter 4.

Chapter 4

Simulation

We used four workloads in the SPEC CINT2006 benchmark suites [11]. We performed profiling and baseline experiments on the Intel Xeon E5630 server equipped with 16GB of memory.

4.1 Simulation Methodology

The simulations consist of the following phases: the profiling phase and the object placement phase.

Profiling Phase To identify the number and memory access patterns of all objects in an application, we first profile each application statically and dynamically. With static profiling, the CAMP profiler detects all possible contexts in the application, assigns unique context IDs, and generates its own context tree. After detecting contexts and assigning IDs, dynamic profiling is performed to identify the memory access patterns of the application.

Object Placement Phase Using the profiling result of the application, we estimate the effects of object placement on different types of memory. We first allocate all objects in each application to the homogeneous memory system which only composed of DRAM or PCM. Then we estimate the memory access latency and the energy consumption for all objects in each application according to the equations introduced in the Section 3.2 and 3.3.

We performed object placement simulation to verify the effects of various thresholds and the effects of various memory composition ratios within the heterogeneous memory system. The results of the simulation under various situations are shown in the following sections. We take the simulation results in the DRAM-only system as the baseline, and all other simulation results are normalized to the baseline. We first provide profiling results for each workload and then show simulation results for different memory systems in terms of memory access latency and energy consumption.

4.2 Program Profiling Results

Each workload used in the simulation has different characteristics. For example, 401.bzip2 allocates a small number of objects during runtime, but the sum of the sizes of all allocated objects is the largest of all workloads. However, our modeling method could reflect the various characteristics of the workloads since placement model considers the size of the object as well as the access patterns of all objects.

Table 4.1 summarizes the characteristics of objects allocated during each workload runtime. For efficient profiling in terms of time and memory space,

Benchmark	# of Object	# of Load	# of Store	Total Object Size
401.bzip2	12	9661190K	5942637K	205.74 MB
433.milc	80	2285696K	511474K	18.69 MB
456.hmmer	36	1818391K	24623K	1.42 MB
462.libquantum	10	26735K	8762K	6.32 MB

Table 4.1: Object Profiling Details

(K means Thousands)

we adjusted the input arguments for each workload in the benchmark suite.

With the object profiling results and the modeling methods described in Chapter 3, we simulate object placement to different types of memory. Since our modeling method reflects runtime behavior of accessing memory in different workloads, each workload has its own threshold for placing objects on heterogeneous memory systems. Objects that cause a big amount of store instructions are placed in DRAM to reduce the memory access latency and power overhead. Objects that are allocated at a large size but do not issue a store instruction are placed on PCM because there is no overhead due to the write operation.

Benchmark		# of Object	# of Load	# of Store	Total Object Size
401.bzip2	DRAM	6	9660994K	5942571K	3.81 MB
	PCM	6	196K	65K	201.93 MB
433.milc	DRAM	39	2148786K	494774K	7.34 MB
	PCM	41	136910K	16700K	11.35 MB
456.hmmer	DRAM	18	1818391K	24623K	1.42 MB
	PCM	18	686	31	0.00 MB
462.libquantum	DRAM	4	25154K	8761K	4.13 MB
	PCM	6	1581K	173	2.20 MB

Table 4.2: Object Placement Details (threshold = median value)

(K means Thousands)

Table 4.2 shows an example of object placement for each workload when we set threshold as the median value.

4.3 Simulation of Latency

The Figure 4.1 shows the simulation results of memory access latency according to object placement in different memory systems. As can be seen, the latency difference between DRAM and PCM is from 3.13 times to 20.59 times. The simulation results show that object placement with proposed method could achieve the similar latency on heterogeneous memory system to that of the DRAM-only system. The reason is that some objects in those workloads are never issuing write instructions. On the other hand, the simulation result for the 433.milc workload is 1.37 times the latency of the DRAM-only system, slightly higher latency difference than other workloads. This is because the amount of the write access to PCM is higher than other workloads. We expected that further adjustment of the threshold to a value other than the median could achieve performance improvements for the 433.milc, too. To verify our assumption, we performed another simulation by setting the threshold to zero, and the results are shown together in the Figure 4.1. For the 433.milc and the 462.libquantum workloads, threshold adjustment improves performance by reducing the number of writes to PCM.

We also simulated object placement under various memory composition ratios within the system. The Figure 4.2 presents the simulation results of memory access latency under various DRAM size limitations. For the 401.bzip2 and the 456.hmmer workloads, memory configuration does not affect latency because small DRAMs are sufficient to allocate objects that issue many write instructions. However, the latency increases for the 462.libquantum workload

when changing DRAM ratio from 50% to 25%, due to an object which issues the largest number of write instructions. The object is allocated to DRAM when its ratio is set to 50% but moved to PCM when DRAM ratio is changed to 25%, which causes latency increment. The simulation results of 433.milc workload show latencies that meet our expectations. The 433.milc workload allocates many objects at various sizes during runtime, allowing for fine-grained object placement in various memory configurations.

4.4 Simulation of Energy Consumption

The Figure 4.3 shows the simulation results of total energy consumption based on object placement in different memory systems. The main reason for the difference in energy consumption between DRAM and PCM is the high write energy of the PCM. From the result of object placement modeling, objects that have few write instructions are placed on PCM. Different to the latency modeling, energy consumption modeling also considers the size of an object. Therefore, placing objects on heterogeneous memory system could reduce energy consumption differences to nearly zero on all workloads in the benchmark, even with different threshold values. However, energy consumption could be varied with different memory configurations. As shown in the Figure 4.4, the performance gap is similar to latencies.

4.5 Simulation of Idle Power Consumption

The Figure 4.5 shows simulation results of idle power consumption of memory systems based on the object allocation in different memory systems. Idle power consumption is one of the most attractive characteristics of PCM. The simulation is based on the assumption that all objects allocated to the application are

maintained during application runtime. In other words, we simulate the maximum idle power consumption of memory systems with different configurations. Idle power consumption is proportional to the total object size allocated to different memory during application runtime. Idle power consumption increases corresponding to the allocation size when objects are placed to heterogeneous memory systems compared with the PCM-only system. However, the 401.bzip2 workload reduces idle power consumption when using heterogeneous memory system. This abnormal result is caused by the object characteristics allocated during workload runtime. The 401.bzip2 workload allocates objects of large size during runtime, but there are near zero reads and writes instructions to those objects. The object placement result shown in table 4.2 demonstrates this.

We also simulated idle power consumption of memory system under various memory configurations as in previous Sections. The simulation results are presented in the Figure 4.6. Unlike latency and energy consumption results, the idle power consumption of the memory system differs greatly depending on the memory configurations. Since the idle power difference between DRAM and PCM is 100 times according to the table 2.1, the allocation of one object to another memory has a significant impact on overall power consumption.

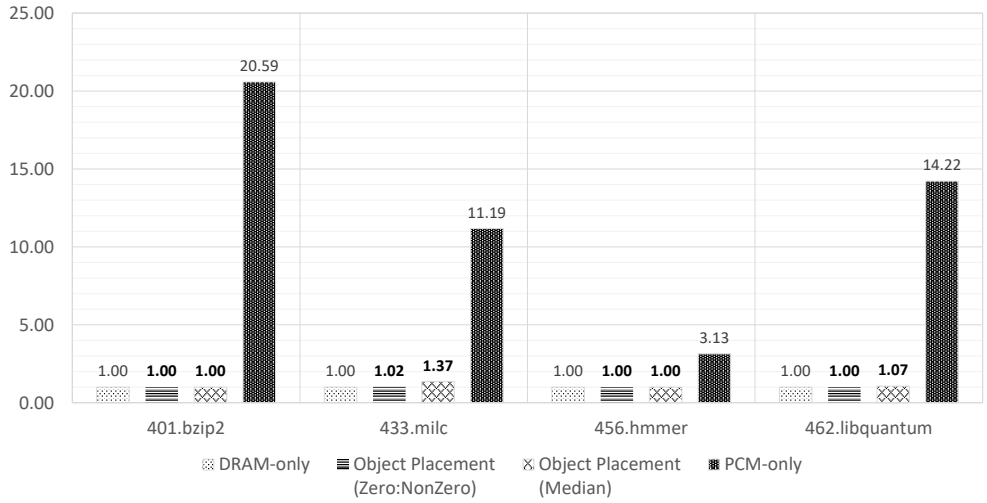


Figure 4.1: Simulation of Object Load/Store Latency with various threshold

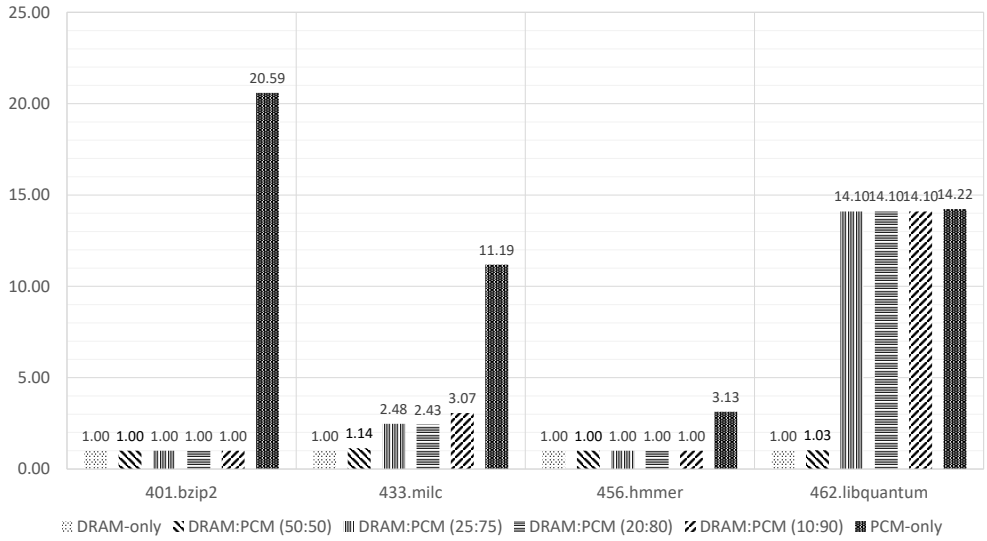


Figure 4.2: Simulation of Object Load/Store Latency
with various memory composition ratios

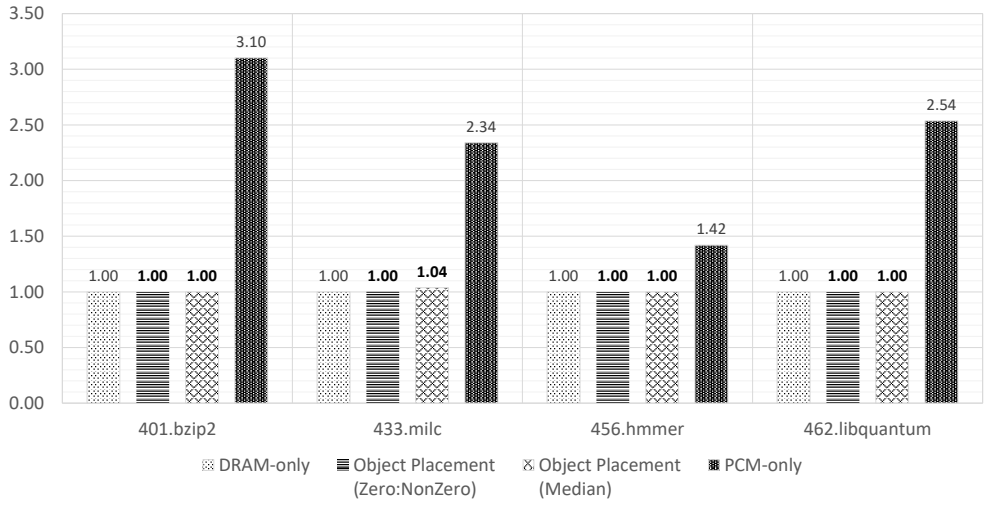


Figure 4.3: Simulation of Object Load/Store Energy Consumption with various threshold

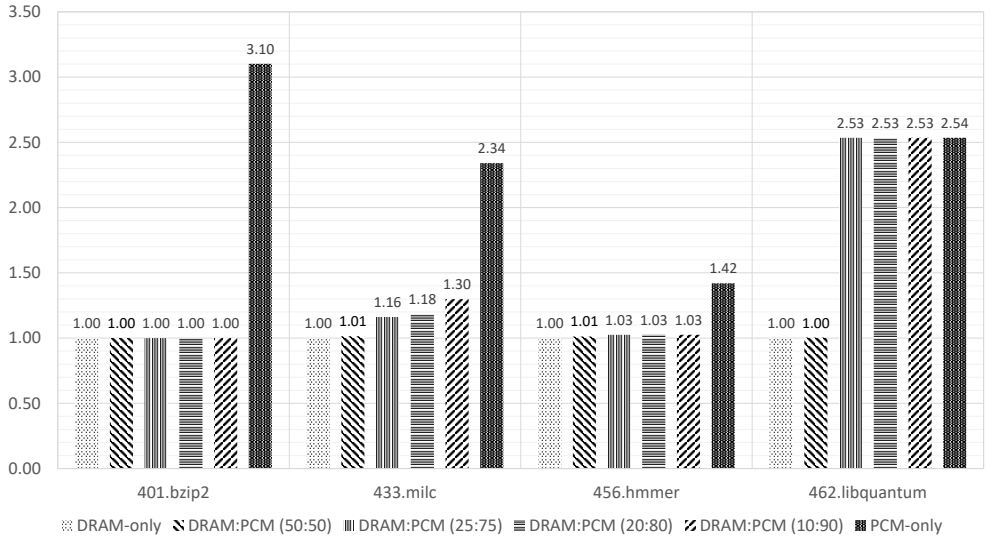


Figure 4.4: Simulation of Object Load/Store Energy Consumption with various memory composition ratios

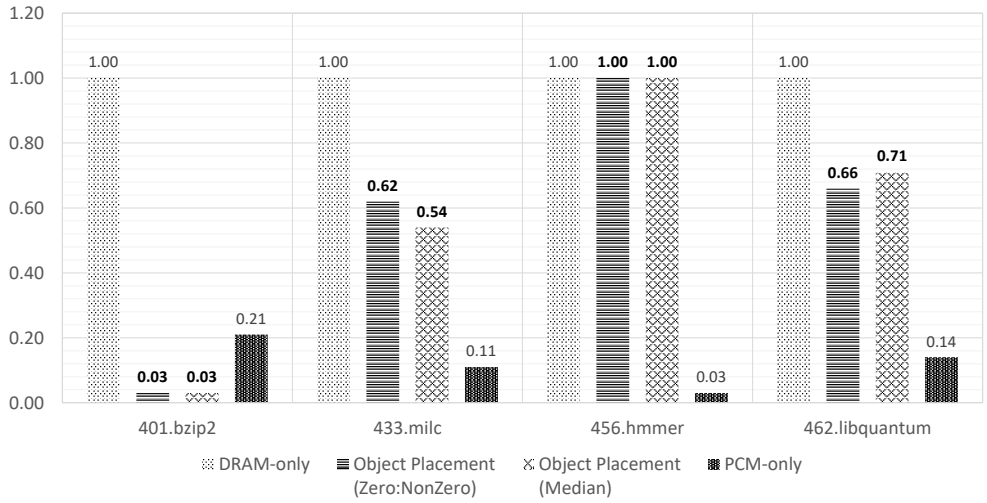


Figure 4.5: Simulation of Idle Power Consumption with various threshold

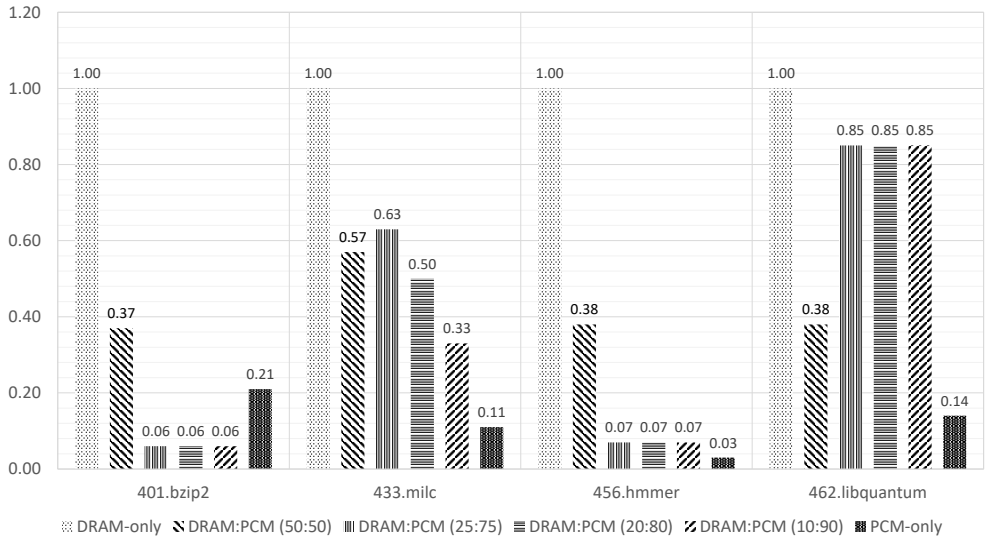


Figure 4.6: Simulation of Idle Power Consumption
with various memory composition ratios

Chapter 5

Conclusion

In this paper, we propose an object placement modeling method on heterogeneous memory system with object profiling with full program context information. The compiler uses memory access patterns and the size of each object to dynamically determine the appropriate memory for each object to be placed. Proper placement of objects on heterogeneous memory systems could reduce system-wide energy consumption while maintaining DRAM-like application latency. By deciding object allocation at the compiler-level, applications could use heterogeneous memory systems transparently. Furthermore, since context-aware memory profiler provides memory access pattern information at object-granularity, allowing the system to manage memory in more fine-grained fashion.

Bibliography

[1] P. Chi, W. C. Lee and Y. Xie., "Adapting B^+ -Tree for Emerging Nonvolatile Memory-Based Main Memory", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, No. 9, 2016.

[2] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y., "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology", in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, 2009.

[3] P. Chi, W.-C. Lee, and Y. Xie., "Making B^+-Tree Efficient in PCM-Based Main Memory", in *Proceedings of ISLPED'14*, 2014.

[4] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan., "Data Tiering in Heterogeneous Memory Systems", in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[5] Kai Wu, Yingchao Huang, and Dong Li., "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory", in *Proceedings of SC'17*, 2017.

[6] J. Kim, “Context -Aware Memory Dependence Profiling”, Master’s Thesis, CSE Department, POSTECH, 2017.

[7] E. Doller., Phase change memory and its impacts on memory hierarchy, http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[8] Soyoon Lee, and Hyokyung Bahn, and Sam H. Noh., “CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures”, in *IEEE Transactions on Computers*, Vol. 63. No. 9, 2014.

[9] Ramos, Luiz E. and Gorbatov, Eugene and Bianchini, Ricardo., “Page Placement in Hybrid Memory Systems”, in *Proceedings of the International Conference on Supercomputing (ICS ’11)*, 2011.

[10] C. Lattner and V. Adve., “Llvm: A compilation framework for lifelong program analysis & transformation”, in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[11] Standard Performance Evaluation Corporation, http://www.spec.org.

초록

PCM 은 대용량 및 낮은 소비 전력이라는 장점을 가지는 NVM 기술 중 하나이다. 그러나 상대적으로 느린 메모리 접근 시간은 PCM 을 DRAM 과 같은 메인 메모리로 사용하는 데에 있어 주요 걸림돌 중 하나로 작용한다. DRAM 의 빠른 메모리 접근 시간과 PCM 의 낮은 소비 전력이라는 장점을 극대화하기 위해, DRAM 과 PCM 을 활용한 이기종 메모리 시스템을 구성하는 연구들이 최근 진행되어 왔다. 이러한 이기종 메모리 시스템을 사용함에 있어 주요 문제점 중 하나는 적절한 유형의 메모리에 데이터를 배치하는 것이다. 본 논문에서는 이기종 메모리 시스템에서 객체 단위의 데이터를 적절한 메모리에 배치하는 방법을 제안한다. 컨텍스트를 인지하는 객체 프로파일링 정보를 이용하여 객체의 메모리 접근 패턴을 동적으로 감지하고 객체를 배치할 적절한 메모리를 결정한다. 선정된 4 개의 SPEC 벤치마크 프로그램에 대한 메모리 접근 시간 및 소비 전력량 시뮬레이션 결과를 통해 제안된 방법의 효과를 입증한다.

주요어: 객체 배치, 이기종 메모리 시스템, 컨텍스트를 인지하는 객체 프로파일링
학번: 2016-21198

M.S. THESIS

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

FEBRUARY 2018

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

M.S. THESIS

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

FEBRUARY 2018

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hwajung Kim

Simulating Object Placement in Heterogeneous Memory System using Context-Aware Object Profiling Information

컨텍스트를 인지하는 객체 프로파일링 정보를 이용한 이기종 메모리 시스템에서의 객체 배치 시뮬레이션

지도교수 염헌영

이 논문을 공학석사 학위논문으로 제출함

2017 년 11 월

서울대학교 대학원

컴퓨터 공학부

김화정

김화정의 공학석사 학위논문을 인준함

2018 년 01 월

위 원 장	엄현상	(인)
부위원장	염헌영	(인)
위 원	전병곤	(인)

Abstract

Phase change memory (PCM) is one of the promising non-volatile memory (NVM) technologies since it provides both high capacity and low idle power consumption. However, relatively slow access latency is one of the major challenges in using PCM as main memory. Therefore, in recent researches, it is attempting to construct heterogeneous memory systems by combining such NVM with DRAM. One of the major problems with using those systems is placing the data in the appropriate type of memory. In this paper, we propose an object placement method to address data placement problem in heterogeneous memory systems. With context-aware object profile information, we could dynamically detect memory access patterns of objects and determine the proper memory to place the objects on. We demonstrate the effectiveness of the proposed method by simulating memory access latency and energy consumption using the four selected workloads of the SPEC benchmark.

Keywords: Object Placement, Heterogeneous Memory System, Context-Aware Object Profiling

Student Number: 2016-21198

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Phase change memory (PCM) is one of the emerging non-volatile memory (NVM) devices. PCM has a longer access latency than DRAM and consumes much energy to read and write data [1, 2, 3]. Despite its limitations in latency and bandwidth, it still offers appealing attributes. Its density is much higher than DRAM and provides near-zero idle power consumption. Thus, several studies have been done to construct heterogeneous memory systems to overcome performance overhead and take advantage of NVM [4, 5].

S.R. Dulloor et al. [4] proposed data classification and tiering techniques under hybrid memory architecture. Researchers in [4] used an offline profiling tool (PIN) to understand access patterns of different data structures of applications. On the other hand, Kai Wu et al. [5] introduced Unimem to dynamically place objects on heterogeneous memory systems. Unimem focuses on memory access patterns of applications that operate in iterative structures. It manages data placement based on runtime profiling and performance models and targets high-performance computing (HPC) systems.

In this paper, we propose an object placement modeling method using the knowledge of access patterns of objects to the memory. To capture memory access patterns of objects allocated by an application, we use CAMP [6] framework. It is a framework that provides context-aware memory dependency information, and we will provide a more detailed description in Section 2.2. Using CAMP framework, we could collect memory access information of each object and allocate to appropriate memory at the compiler-level. We propose a modeling method that demonstrates the effect of placing objects on various memory systems in terms of latency and energy consumption, depending on the memory access patterns of the objects allocated by the application. To verify the effect of object placement, several simulations were performed assuming different memory systems. Simulation results show that appropriate objects placement in a heterogeneous memory system could achieve a DRAM-like latency while minimizing the overall energy consumption of the system.

Chapter 2

Background and Motivation

This chapter briefly describes the structure of the heterogeneous memory system used in this paper. Also, we introduce the profiling tool and methods that are used to capture the behavior of objects allocated to applications during runtime.

2.1 Heterogeneous Memory Systems

Different to DRAM, PCM has asymmetric read and write characteristics, as has been shown in previous studies [1, 2, 3]. Table 2.1 compares latency and energy consumption for DRAM and PCM access [3, 7]. Using PCM instead of DRAM requires up to 50 times the latency and 6 times the energy for a single write operation. However, PCM has some attractive characteristics in spite of such performance limitations. First of all, PCM maintains data persistently even in the event of a power failure. PCM also has a higher density than DRAM and has near-zero idle power consumption. Therefore, several studies have been conducted to compose heterogeneous memory systems utilizing PCM as well as

a DRAM [8, 9].

	DRAM	PCM
Page Size	64 B	64 B
Read Latency	20-50 ns	$\sim$ 50 ns
Write Latency	20-50 ns	$\sim$ 1 μs
Read Energy	0.8 J/GB	1 J/GB
Write Energy	1.2 J/GB	6 J/GB
Idle Power	$\sim$ 100 mW/GB	$\sim$ 1 mW/GB

Table 2.1: Latency and Energy Consumption Comparison between DRAM and PCM access

2.2 Context-Aware Memory Profiling

The CAMP framework is a Context-Aware Memory Profiling framework that traces program memory dependencies using full context information of a program, such as a call site stack and loop nesting levels. The CAMP statically detects all possible contexts in the program and assigns a unique ID to each context. It also generates a static context tree for the program, as shown in the Figure 2.1. Figure 2.1b shows the context tree for the example program shown in Figure 2.1a.

2.3 Object Profiling and Placement

We use the CAMP framework [6] to understand memory access patterns of applications. The CAMP framework is implemented on top of the LLVM compiler infrastructure [10]. With the LLVM infrastructure, it is easy to apply compiler-level optimization features. The original CAMP framework provides profiling results for all possible load and store dependencies of objects allocated by the

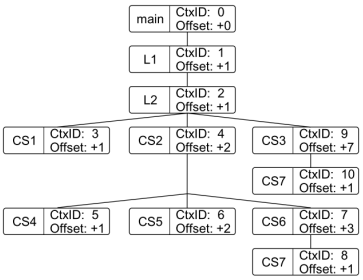
application. We added a few lines of code to the CAMP framework to identify all objects access patterns in the application. We also implemented additional optimization pass in the CAMP framework to dynamically place objects on the heterogeneous memory system at the compiler-level.

We named the additional pass as ctx-obj-placement. The pass automatically allocates objects to the heterogeneous memory system, which are transparent to the application. In the pass, it first reads the object access pattern profiling results generated by the previous dynamic profiling execution. The pass then calculates the amount of write access to each object and determines the threshold at which to place the object on the heterogeneous memory system. During the runtime of the application, the pass automatically catches memory allocation requests from the application and allocates space in the appropriate memory.

Figure 2.2 summarizes the application profiling process of the CAMP framework.

```
1  int getValue(Node *n) {
2    return n->value;                  // LD1
3  }
4
5  void setValue(Node *n, int v) {
6    if(isValid(v))                    // CS7
7      n->value = v;                   // ST1
8  }
9
10 int work(Node *n) {
11   int v1 = getValue(n);             // CS4
12   int v2 = update(v1);              // CS5
13   setValue(n, v2);                  // CS6
14   return v2;
15 }
16
17 void main() {
18   for(int t = 0; t < T; t++) {      // L1
19     for(int i = 0; i < N; i++) {    // L2
20       int s = getValue(sum[t]);     // CS1
21       int v = work(nodes[i]);       // CS2
22       s = s + v;                    // ADD
23       setValue(sum[t], s);          // CS3
24     }
25   }
26 }
```

(a) Example Program



(b) Context Tree for the Example Program

Figure 2.1: Context-Aware Memory Dependence Profiling Example

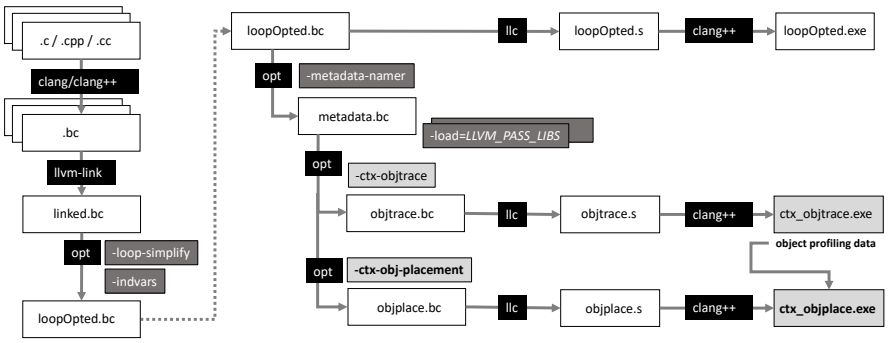


Figure 2.2: Context-Aware Object Trace using LLVM

Chapter 3

Object Placement Modeling

In this chapter, we explain the modeling method to simulate the performance of the application in terms of latency and energy consumption on the heterogeneous memory system. Depending on the modeling method, a detailed description of how to determine the appropriate memory to allocate for each object in the application is provided at the end of this chapter.

3.1 Basic Assumptions

To simplify the object placement problem, we establish following assumptions.

Assumption 1. *Caching effects on objects are the same in various memory systems.*

Assumption 2. *The difference in access latency between DRAM and PCM is maximum. That is, we set the DRAM read and write to 20 ns and the PCM to 50 ns and 1 μs.*

Assumption 3. *The application latency is proportional to the memory access latency of objects referenced during runtime. Thus, the performance comparison between different types of memory is based on the memory access pattern of all objects allocated by the application.*

3.2 Latency Modeling

From the dynamic profiling results of each object, we can figure out how many times each object is loaded and stored while the application is running. With the size of the allocated object, latency prediction modeling is as follows. The Equation 3.1 and the Equation 3.2 represent the load and store latency for each object allocated to a heterogeneous memory system.

The load latency of an object ($LatLD_{obj}$) is determined by the read latency of DRAM ($LatLD_{DRAM}$), the number of load instructions for DRAM of object ($NumLD_{DRAM}$), the read latency of PCM ($LatLD_{PCM}$), and the number of load instructions for PCM of object ($NumLD_{PCM}$). Likewise, the store latency of an object ($LatST_{obj}$) is determined by the write latency of DRAM ($LatST_{DRAM}$), the number of store instructions for DRAM of object ($NumST_{DRAM}$), the write latency of PCM ($LatST_{PCM}$), and the number of store instructions for PCM of object ($NumST_{PCM}$).

$$LatLD_{obj} = LatLD_{DRAM} \times NumLD_{DRAM} + LatLD_{PCM} \times NumLD_{PCM} \tag{3.1}$$

$$LatST_{obj} = LatST_{DRAM} \times NumST_{DRAM} + LatST_{PCM} \times NumST_{PCM} \tag{3.2}$$

The Equation 3.3 represents the latency of an application. It is the sum of load and store latency of each object allocated while the application is running.

$$Latency_{application} = \sum_{obj}(LatLD_{obj} + LatST_{obj}) \tag{3.3}$$

3.3 Energy Consumption Modeling

Energy consumption modeling considers the size of each object allocated to the heterogeneous memory system, as opposed to the latency prediction modeling introduced in Section 3.2. The Equation 3.4 and the Equation 3.5 represent the load and store energy consumption of each object allocated to the heterogeneous memory system.

The load energy consumption of an object ($EnergyLD_{obj}$) is determined by the load energy consumption of DRAM ($EnergyLD_{DRAM}$), the number of load instructions for DRAM of the object ($NumLD_{DRAM}$), the size of the object allocated on DRAM ($Size_{DRAM}$), the load energy consumption of PCM ($EnergyLD_{PCM}$), the number of load instructions for PCM of the object ($NumLD_{PCM}$), and the size of the object allocated on PCM ($Size_{PCM}$). Likewise, the store energy consumption of an object ($EnergyST_{obj}$) is determined by the store energy consumption of DRAM ($EnergyST_{DRAM}$), the number of store instructions for DRAM of the object ($NumST_{DRAM}$), the size of the object allocated on DRAM ($Size_{DRAM}$), the store energy consumption of PCM ($EnergyST_{PCM}$), the number of store instructions for PCM of the object ($NumST_{PCM}$), and the size of the object allocated on PCM ($Size_{PCM}$).

$$\begin{aligned} EnergyLD_{obj} = & EnergyLD_{DRAM} \times NumLD_{DRAM} \times Size_{DRAM} \\ & + EnergyLD_{PCM} \times NumLD_{PCM} \times Size_{PCM} \end{aligned} \tag{3.4}$$

$$\begin{aligned} EnergyST_{obj} = & EnergyST_{DRAM} \times NumST_{DRAM} \times Size_{DRAM} \\ & + EnergyST_{PCM} \times NumST_{PCM} \times Size_{PCM} \end{aligned} \tag{3.5}$$

The Equation 3.6 represents the energy consumption of an application. It is the sum of $EnergyLD_{obj}$ and $EnergyST_{obj}$ for all objects allocated while the application is running.

$$Energy_{application} = \sum_{obj} (EnergyLD_{obj} + EnergyST_{obj}) \tag{3.6}$$

3.4 Idle Power Consumption Modeling

We also simulated the idle power consumption of the memory system while an application is running, based on the simulated latency of the application and the total object size allocated during application runtime. The Equation 3.7 represents the idle power consumption of the memory system.

In the equation, $Latency_{application}$ is calculated from the Equation 3.3. $Size_{DRAM}$ and $Size_{PCM}$ are the total object size allocated to DRAM and PCM while the application is running.

$$\begin{aligned} IdlePower_{memory} = & Latency_{application} \times IdlePower_{DRAM} \times Size_{DRAM} \\ & + Latency_{application} \times IdlePower_{PCM} \times Size_{PCM} \end{aligned} \tag{3.7}$$

3.5 Object Placement Decision

As mentioned in previous chapters, the performance difference between DRAM and PCM depends primarily on the total amount of write instructions of the application. Therefore, we determine object placement based on the number and

the size of write instructions for each object. For all objects allocated by the application, we first calculate the total amount of write access to the memory of each object. We then choose proper value as the threshold.

The compiler pass first scans through all objects and compare the total amount of write access of each object to the threshold value during initialization of the application. The object is placed in DRAM when the total amount is greater than the threshold, otherwise, it is placed in PCM. Table 4.2 presents the result of object placement based on profiling information with the median value as the threshold.

Thresholds can vary depending on the characteristics of the application. In the paper, we simulated with two different threshold values, zero and the median. All simulation results are given in Chapter 4.

Chapter 4

Simulation

We used four workloads in the SPEC CINT2006 benchmark suites [11]. We performed profiling and baseline experiments on the Intel Xeon E5630 server equipped with 16GB of memory.

4.1 Simulation Methodology

The simulations consist of the following phases: the profiling phase and the object placement phase.

Profiling Phase To identify the number and memory access patterns of all objects in an application, we first profile each application statically and dynamically. With static profiling, the CAMP profiler detects all possible contexts in the application, assigns unique context IDs, and generates its own context tree. After detecting contexts and assigning IDs, dynamic profiling is performed to identify the memory access patterns of the application.

Object Placement Phase Using the profiling result of the application, we estimate the effects of object placement on different types of memory. We first allocate all objects in each application to the homogeneous memory system which only composed of DRAM or PCM. Then we estimate the memory access latency and the energy consumption for all objects in each application according to the equations introduced in the Section 3.2 and 3.3.

We performed object placement simulation to verify the effects of various thresholds and the effects of various memory composition ratios within the heterogeneous memory system. The results of the simulation under various situations are shown in the following sections. We take the simulation results in the DRAM-only system as the baseline, and all other simulation results are normalized to the baseline. We first provide profiling results for each workload and then show simulation results for different memory systems in terms of memory access latency and energy consumption.

4.2 Program Profiling Results

Each workload used in the simulation has different characteristics. For example, 401.bzip2 allocates a small number of objects during runtime, but the sum of the sizes of all allocated objects is the largest of all workloads. However, our modeling method could reflect the various characteristics of the workloads since placement model considers the size of the object as well as the access patterns of all objects.

Table 4.1 summarizes the characteristics of objects allocated during each workload runtime. For efficient profiling in terms of time and memory space,

Benchmark	# of Object	# of Load	# of Store	Total Object Size
401.bzip2	12	9661190K	5942637K	205.74 MB
433.milc	80	2285696K	511474K	18.69 MB
456.hmmer	36	1818391K	24623K	1.42 MB
462.libquantum	10	26735K	8762K	6.32 MB

Table 4.1: Object Profiling Details

(K means Thousands)

we adjusted the input arguments for each workload in the benchmark suite.

With the object profiling results and the modeling methods described in Chapter 3, we simulate object placement to different types of memory. Since our modeling method reflects runtime behavior of accessing memory in different workloads, each workload has its own threshold for placing objects on heterogeneous memory systems. Objects that cause a big amount of store instructions are placed in DRAM to reduce the memory access latency and power overhead. Objects that are allocated at a large size but do not issue a store instruction are placed on PCM because there is no overhead due to the write operation.

Benchmark		# of Object	# of Load	# of Store	Total Object Size
401.bzip2	DRAM	6	9660994K	5942571K	3.81 MB
	PCM	6	196K	65K	201.93 MB
433.milc	DRAM	39	2148786K	494774K	7.34 MB
	PCM	41	136910K	16700K	11.35 MB
456.hmmer	DRAM	18	1818391K	24623K	1.42 MB
	PCM	18	686	31	0.00 MB
462.libquantum	DRAM	4	25154K	8761K	4.13 MB
	PCM	6	1581K	173	2.20 MB

Table 4.2: Object Placement Details (threshold = median value)

(K means Thousands)

Table 4.2 shows an example of object placement for each workload when we set threshold as the median value.

4.3 Simulation of Latency

The Figure 4.1 shows the simulation results of memory access latency according to object placement in different memory systems. As can be seen, the latency difference between DRAM and PCM is from 3.13 times to 20.59 times. The simulation results show that object placement with proposed method could achieve the similar latency on heterogeneous memory system to that of the DRAM-only system. The reason is that some objects in those workloads are never issuing write instructions. On the other hand, the simulation result for the 433.milc workload is 1.37 times the latency of the DRAM-only system, slightly higher latency difference than other workloads. This is because the amount of the write access to PCM is higher than other workloads. We expected that further adjustment of the threshold to a value other than the median could achieve performance improvements for the 433.milc, too. To verify our assumption, we performed another simulation by setting the threshold to zero, and the results are shown together in the Figure 4.1. For the 433.milc and the 462.libquantum workloads, threshold adjustment improves performance by reducing the number of writes to PCM.

We also simulated object placement under various memory composition ratios within the system. The Figure 4.2 presents the simulation results of memory access latency under various DRAM size limitations. For the 401.bzip2 and the 456.hmmer workloads, memory configuration does not affect latency because small DRAMs are sufficient to allocate objects that issue many write instructions. However, the latency increases for the 462.libquantum workload

when changing DRAM ratio from 50% to 25%, due to an object which issues the largest number of write instructions. The object is allocated to DRAM when its ratio is set to 50% but moved to PCM when DRAM ratio is changed to 25%, which causes latency increment. The simulation results of 433.milc workload show latencies that meet our expectations. The 433.milc workload allocates many objects at various sizes during runtime, allowing for fine-grained object placement in various memory configurations.

4.4 Simulation of Energy Consumption

The Figure 4.3 shows the simulation results of total energy consumption based on object placement in different memory systems. The main reason for the difference in energy consumption between DRAM and PCM is the high write energy of the PCM. From the result of object placement modeling, objects that have few write instructions are placed on PCM. Different to the latency modeling, energy consumption modeling also considers the size of an object. Therefore, placing objects on heterogeneous memory system could reduce energy consumption differences to nearly zero on all workloads in the benchmark, even with different threshold values. However, energy consumption could be varied with different memory configurations. As shown in the Figure 4.4, the performance gap is similar to latencies.

4.5 Simulation of Idle Power Consumption

The Figure 4.5 shows simulation results of idle power consumption of memory systems based on the object allocation in different memory systems. Idle power consumption is one of the most attractive characteristics of PCM. The simulation is based on the assumption that all objects allocated to the application are

maintained during application runtime. In other words, we simulate the maximum idle power consumption of memory systems with different configurations. Idle power consumption is proportional to the total object size allocated to different memory during application runtime. Idle power consumption increases corresponding to the allocation size when objects are placed to heterogeneous memory systems compared with the PCM-only system. However, the 401.bzip2 workload reduces idle power consumption when using heterogeneous memory system. This abnormal result is caused by the object characteristics allocated during workload runtime. The 401.bzip2 workload allocates objects of large size during runtime, but there are near zero reads and writes instructions to those objects. The object placement result shown in table 4.2 demonstrates this.

We also simulated idle power consumption of memory system under various memory configurations as in previous Sections. The simulation results are presented in the Figure 4.6. Unlike latency and energy consumption results, the idle power consumption of the memory system differs greatly depending on the memory configurations. Since the idle power difference between DRAM and PCM is 100 times according to the table 2.1, the allocation of one object to another memory has a significant impact on overall power consumption.

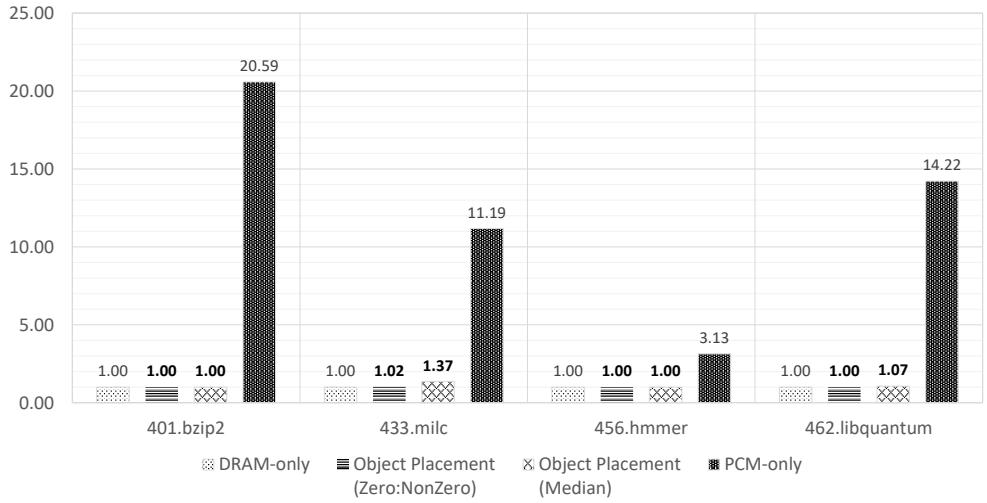


Figure 4.1: Simulation of Object Load/Store Latency with various threshold

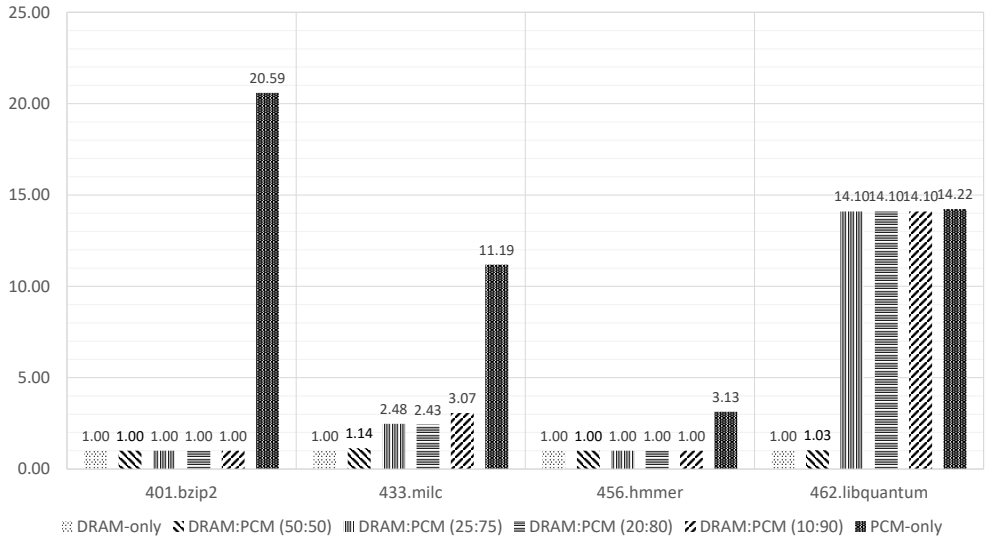


Figure 4.2: Simulation of Object Load/Store Latency
with various memory composition ratios

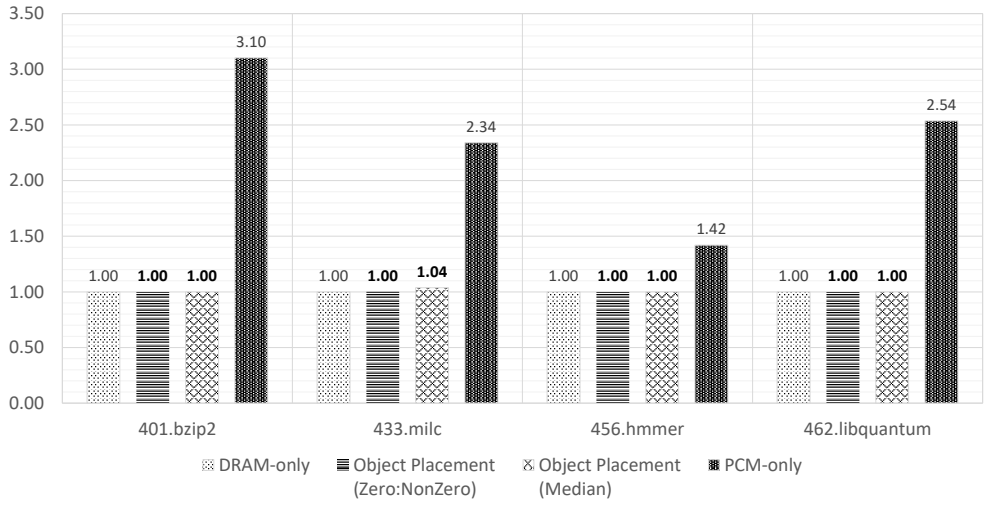


Figure 4.3: Simulation of Object Load/Store Energy Consumption with various threshold

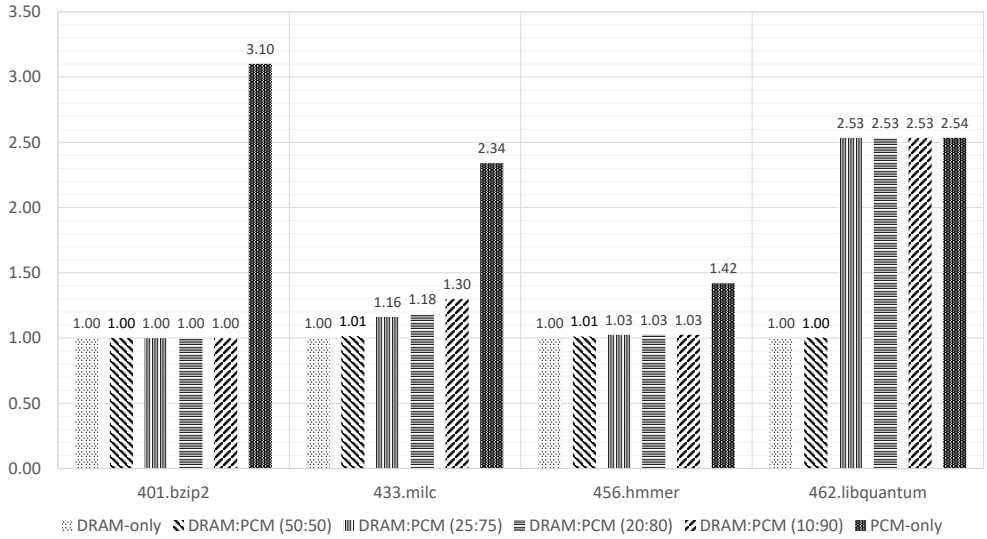


Figure 4.4: Simulation of Object Load/Store Energy Consumption with various memory composition ratios

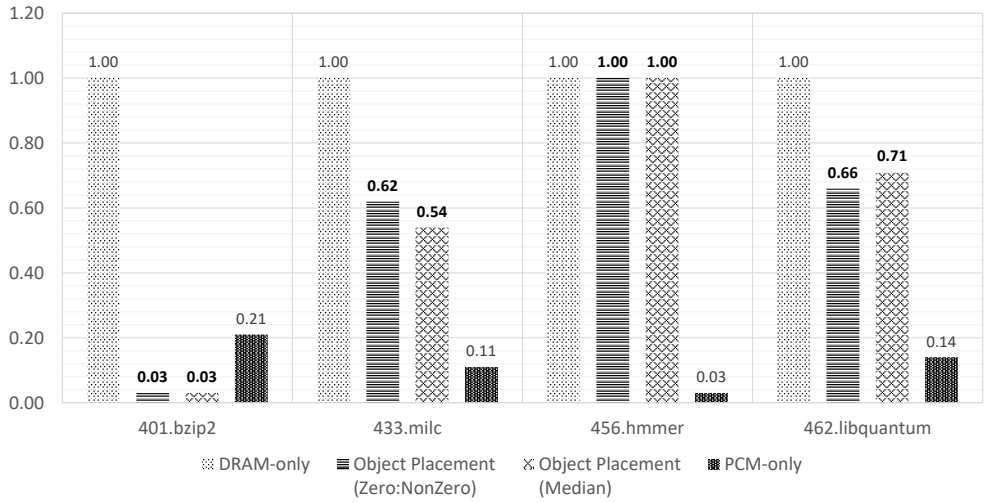


Figure 4.5: Simulation of Idle Power Consumption with various threshold

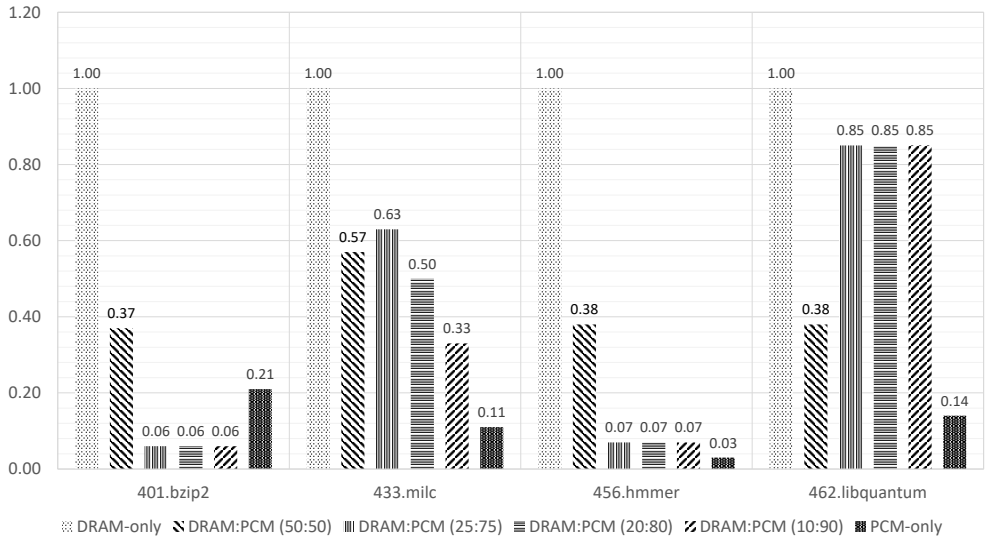


Figure 4.6: Simulation of Idle Power Consumption
with various memory composition ratios

Chapter 5

Conclusion

In this paper, we propose an object placement modeling method on heterogeneous memory system with object profiling with full program context information. The compiler uses memory access patterns and the size of each object to dynamically determine the appropriate memory for each object to be placed. Proper placement of objects on heterogeneous memory systems could reduce system-wide energy consumption while maintaining DRAM-like application latency. By deciding object allocation at the compiler-level, applications could use heterogeneous memory systems transparently. Furthermore, since context-aware memory profiler provides memory access pattern information at object-granularity, allowing the system to manage memory in more fine-grained fashion.

Bibliography

[1] P. Chi, W. C. Lee and Y. Xie., "Adapting B^+ -Tree for Emerging Nonvolatile Memory-Based Main Memory", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, No. 9, 2016.

[2] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y., "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology", in *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, 2009.

[3] P. Chi, W.-C. Lee, and Y. Xie., "Making B^+-Tree Efficient in PCM-Based Main Memory", in *Proceedings of ISLPED'14*, 2014.

[4] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan., "Data Tiering in Heterogeneous Memory Systems", in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[5] Kai Wu, Yingchao Huang, and Dong Li., "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory", in *Proceedings of SC'17*, 2017.

[6] J. Kim, “Context -Aware Memory Dependence Profiling”, Master’s Thesis, CSE Department, POSTECH, 2017.

[7] E. Doller., Phase change memory and its impacts on memory hierarchy, http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[8] Soyoon Lee, and Hyokyung Bahn, and Sam H. Noh., “CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures”, in *IEEE Transactions on Computers*, Vol. 63. No. 9, 2014.

[9] Ramos, Luiz E. and Gorbatov, Eugene and Bianchini, Ricardo., “Page Placement in Hybrid Memory Systems”, in *Proceedings of the International Conference on Supercomputing (ICS ’11)*, 2011.

[10] C. Lattner and V. Adve., “Llvm: A compilation framework for lifelong program analysis & transformation”, in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[11] Standard Performance Evaluation Corporation, http://www.spec.org.

초록

PCM 은 대용량 및 낮은 소비 전력이라는 장점을 가지는 NVM 기술 중 하나이다. 그러나 상대적으로 느린 메모리 접근 시간은 PCM 을 DRAM 과 같은 메인 메모리로 사용하는 데에 있어 주요 걸림돌 중 하나로 작용한다. DRAM 의 빠른 메모리 접근 시간과 PCM 의 낮은 소비 전력이라는 장점을 극대화하기 위해, DRAM 과 PCM 을 활용한 이기종 메모리 시스템을 구성하는 연구들이 최근 진행되어 왔다. 이러한 이기종 메모리 시스템을 사용함에 있어 주요 문제점 중 하나는 적절한 유형의 메모리에 데이터를 배치하는 것이다. 본 논문에서는 이기종 메모리 시스템에서 객체 단위의 데이터를 적절한 메모리에 배치하는 방법을 제안한다. 컨텍스트를 인지하는 객체 프로파일링 정보를 이용하여 객체의 메모리 접근 패턴을 동적으로 감지하고 객체를 배치할 적절한 메모리를 결정한다. 선정된 4 개의 SPEC 벤치마크 프로그램에 대한 메모리 접근 시간 및 소비 전력량 시뮬레이션 결과를 통해 제안된 방법의 효과를 입증한다.

주요어: 객체 배치, 이기종 메모리 시스템, 컨텍스트를 인지하는 객체 프로파일링
학번: 2016-21198