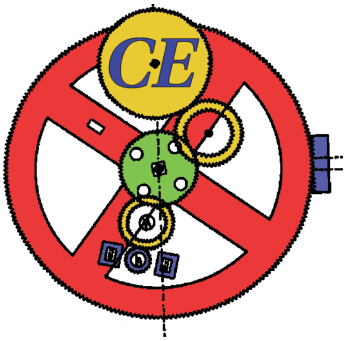

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2015

MSc THESIS

Design space exploration for a Local Object Store

Alexis Ampntel-Kanter-Oikonomou

CE-MS-2015-12

Abstract

Nowadays, modern Integrated Circuit (IC) technology allows processor manufacturers to produce complex designs with up to a few billions of transistors. Technology limitations and the end of voltage and frequency scaling forced computer architecture to multicore designs and more specialized solutions on hardware. These technology trends increased memory bandwidth pressure, exposing modern computer systems to bandwidth limitations. Therefore, a large and increasing fraction of the area is occupied to efficiently manage the available memory bandwidth, especially the off-chip bandwidth. In addition, the failure of Dennard Scaling along with the end of multicore scaling gave rise to the Dark Silicon era, in which the percentage of transistors in an IC that can be simultaneously powered-on is decreasing. Thus, under a fixed power and thermal budget, decreasing the main memory accesses of a processor relieves operational costs and rises the potential for increased computing performance.
In order to tackle this problem, we investigate the possibilities to offload functionalities of the software to hardware due to the drop in the cost of hardware. Based on our observations on modern programming languages, we formulate the hypothesis that *object caching* presents potential to advance the main processor's functionalities in handling objects which are the most commonly used memory structure in modern languages. After formulating the concept of a Local Object Store (LOS), this master thesis presents our progress on developing the framework for our proof of concept. A LOS is expected to decrease the number of main memory references by leveraging the "infant mortality" property of objects in real-life applications. Our proposed design provides support for the basic object manipulation operations while maintaining reference counters for the local objects. Therefore, comparing LOS with a cache memory, it permits the construction and destruction of short-lived objects without polluting the rest of the memory hierarchy. Our framework consists of a Smart Pointer library, two modified tracing tools that allow us to obtain traces from microbenchmarks, and a basic simulator for a Cache-LOS memory unit. Moreover, we develop and trace 3 microbenchmarks and using our framework simulate a 39% average reduction in the main memory accesses for a Cache-LOS memory architecture compared to a cache-only memory configuration. These promising results show the benefits of an integrated Cache-LOS to manipulate object structures.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Design space exploration for a Local Object Store

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Alexis Ampntel-Kanter-Oikonomou
born in Athens, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design space exploration for a Local Object Store

by Alexis Ampntel-Kanter-Oikonomou

Abstract

Nowadays, modern Integrated Circuit (IC) technology allows processor manufacturers to produce complex designs with up to a few billions of transistors. Technology limitations and the end of voltage and frequency scaling forced computer architecture to multicore designs and more specialized solutions on hardware. These technology trends increased memory bandwidth pressure, exposing modern computer systems to bandwidth limitations. Therefore, a large and increasing fraction of the area is occupied to efficiently manage the available memory bandwidth, especially the off-chip bandwidth. In addition, the failure of Dennard Scaling along with the end of multicore scaling gave rise to the Dark Silicon era, in which the percentage of transistors in an IC that can be simultaneously powered-on is decreasing. Thus, under a fixed power and thermal budget, decreasing the main memory accesses of a processor relieves operational costs and rises the potential for increased computing performance.

In order to tackle this problem, we investigate the possibilities to offload functionalities of the software to hardware due to the drop in the cost of hardware. Based on our observations on modern programming languages, we formulate the hypothesis that *object caching* presents potential to advance the main processor's functionalities in handling objects which are the most commonly used memory structure in modern languages. After formulating the concept of a Local Object Store (LOS), this master thesis presents our progress on developing the framework for our proof of concept. A LOS is expected to decrease the number of main memory references by leveraging the "infant mortality" property of objects in real-life applications. Our proposed design provides support for the basic object manipulation operations while maintaining reference counters for the local objects. Therefore, comparing LOS with a cache memory, it permits the construction and destruction of short-lived objects without polluting the rest of the memory hierarchy. Our framework consists of a Smart Pointer library, two modified tracing tools that allow us to obtain traces from microbenchmarks, and a basic simulator for a Cache-LOS memory unit. Moreover, we develop and trace 3 microbenchmarks and using our framework simulate a 39% average reduction in the main memory accesses for a Cache-LOS memory architecture compared to a cache-only memory configuration. These promising results show the benefits of an integrated Cache-LOS to manipulate object structures.

Laboratory : Computer Engineering
Codenumber : CE-MS-2015-12

Committee Members :

Advisor: Zaid Al-Ars, CE, TU Delft

Chairperson: Peter Hofstee, CE, TU Delft

Member: Rene van Leuken, C&S, TU Delft

Dedicated to my family

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

CMS Concurrent-Mark-Sweep

CPU Central Processing Unit

CU Control Unit

G1 Garbage First

GC Garbage Collector

GPU Graphics Processing Unit

HLLC High-Level Language Computer

HPC High Performance Computing

IC Integrated Circuit

I/O Input/Output

JIT Just In Time

JNI Java Native Interface

JVM Java Virtual Machine

JVM TI Java Virtual Machine Tool Interface

LOS Local Object Store

MMU Memory Management Unit

OOP Object-Oriented Programming

OS Operating System

RAII Resource Acquisition Is Initialization

ROI Region Of Interest

TLB Translation Lookaside Buffer

UID Universal Identifier

WCET Worst Case Execution Time

Acknowledgements

Starting my postgraduate studies in TU Delft, I was fascinated by High Performance Computer Architectures. TU Delft taught me a lot on something I am passionately curious about. Therefore, I would like to thank all my professors and the staff of the TU Delft.

Especially, I am really grateful towards my professor Zaid Al-Ars, who helped me expand my knowledge on the topic of modern computer architectures. Zaid gave me the chance to do research on the most interesting research questions I have encountered in my whole academic life. In addition, I would like to thank him also for his trust and belief in my abilities, but also for his personal guidance during the various moments of my life in Delft. Zaid is one of the finest professors that I had the pleasure to be taught by, but also a good friend to me.

Most of all, I will always be filled with gratitude towards Peter Hofstee for his guidance during this master thesis. The completion of my work would not have been possible without him advising me, always in the kindest and humblest way, as well as his immense knowledge and experience in the field. Peter is a true pedagogue and inspiration to me.

Most importantly, I dedicate this thesis to my family and especially to my father for his endless trust and belief in me.

Alexis Ampntel-Kanter-Oikonomou
Delft, The Netherlands
December 10, 2015

Object Tracking Motivation-including prior work overview

1

1.1 Problem Definition

In modern Computer Architectures the processor spends most of its time transferring data between the several units that it controls. This applies to all the data movements performed between the different cache levels and the main memory that has grown to be the dominant cost, but also between register files and the first levels of the cache hierarchy. One of the main consequences is that the primary reason that limits performance is the memory subsystem operational costs. Indeed, the performance of a big class of applications depends on the latency of the data transfers between the different levels of the memory subsystem.

There are more factors that can stall a Central Processing Unit(CPU) and thus degrade the overall performance of the system. Except for the data transfer rate another important factor are the computing resources and Input/Output(IO) interrupts that serve storage and network operations. Both can introduce bottlenecks in a Computer System. In this master thesis we focus only on the memory bottleneck and consider it as the major factor for performance degradation of applications on modern hardware. The memory-CPU performance gap or memory wall is the disparity between the increase in CPU computing speed and the primarily latency between the CPU and the main memory. As far back as the 1980s, this is a well-known problem called the " Memory Wall"[1]. In simple words the processor can serve more than it can ask. Hopefully, Hybrid Memory Cube or another technology will decrease the barrier but it is our expectation that it will not overcome it.

Moreover, another crucial factor in achieving the peak performance in a computer system is efficient programming. In this master thesis one of our primary leading concerns is the programming models used in today's personal computers and servers. We formulate our research questions on the fact that while locality of reference is crucial to performance, standard architectural abstractions and programming languages treat the memory as flat. Thus, we focus on searching for more information on the data locality of applications that a processor could either extract from basic counters to different hardware implementation for caching data on local Caches or Scratchpads. This is a straightforward approach because data movements have a first-order degree of connection with data locality, as they represent the access patterns in an application's memory working set.

In this master thesis, we investigate Java and C++, two of the most popular programming languages as presented in [10], on how their higher-level attributes interface with the memory subsystem. One of the main aspects of efficient programming is modularity. In the context of Object-Oriented Programming(OOP), being modular means

creating objects to structure the source code. In addition, we focus primarily on the problem of how to advance the processor's knowledge on handling objects and on what relations there are between the performance of the memory subsystem and the required operations for handling objects. Generally, our goal is to bring partial control of the memory management of modern programming languages to the hardware.

1.2 Background - Prior work

There is a vast variety of research directions relevant to the context of this master thesis. Beginning from programming models and their implementations, garbage collection algorithms to memory management techniques on software and hardware. Thus, this section presents the research questions along with the literature research that guide us into formulating to the concept of the Local Object Store(LOS).

In the initial phase we evaluate the differences between a Cache-based and a Scratchpad-based Memory Architecture. Except for the different hardware options, we also investigate the relation between the implementation of different object-oriented programming languages and the memory subsystem. Specifically, we are interested in memory management and garbage collection techniques in software and how these functionalities could move from the software to the hardware domain.

1.2.1 Copy or Transfer, a Cache vs Scratchpad battle

A Cache-Memory is a hardware managed unit that exploits the spatial and temporal locality of data movements from and to main memory. The assumption of spatial and temporal locality in real-life applications does not hold always true. This fact translates to energy waste while transferring data when it is not required. On the other side, Scratchpad Memories mitigate for the loss of energy of unnecessary data movements giving to the programmer the option to optimally arrange data movements at runtime. This occurs due to the fact that Scratchpad Memories are software managed and use a separate memory space from the virtual address space that data in the caches is mapped to. The design of a Scratchpad Memory does not require tag checking and compared to a Cache Memory it resolves into a faster unit in terms of accessing its contents.

Scratchpad Memories are used in a number of successful Computer Architectures. IBM's Cell BE Architecture included a Local Store (Scratchpad Memory) and modern Graphics Processing Units(GPUs) utilize Scratchpads. A lot of the research in embedded systems focuses on the benefits of the data locality and time-predictability properties of Scratchpad Memories, thus resulting in better Worst Case Execution Time compared to the same embedded applications running under a Cache-based Memory hierarchy [19]. Also in [21] a comparison between Cache line locking, an alternative scheme to keeping data local, and Scratchpad Memories is presented for hard-real time systems. In addition, the research of [22] suggests the benefits of ScratchPads for time-predictable multicore computing.

The High Performance Computing (HPC) research community also shows interest for the energy and space benefits of using Scratchpads. In [20] the case of Cache only,

Scratchpad only and Hybrid Memory subsystems are evaluated on HPC applications through simulations resulting into less data transfers.

Moreover, data partitioning compiler techniques have been proposed that take advantage of access patterns for different types of data and dividing them between Cache friendly and Scratchpad friendly allocations. In [23] the researchers propose a compiler technique to map object allocations between a Cache and a Scratchpad(SPM) memory in a Hybrid Memory System. Moreover the researchers in [27], use another framework to partition the data between a SPM and a cache based on reuse distance analysis for the target applications. The LOS presented in this thesis extends the aforementioned mapping techniques by offering a partially automated memory management environment.

Generally, in modern computer architectures, a Scratchpad Memory is included in a processor for its simple design. Thus, under constrained specifications for a processor design its beneficial to use a Scratchpad Memory in order to meet the system area, performance, and power budget. The negative side of such a design choice, would be the increased programming effort for the management of the SPM. For the design of the LOS we try to mitigate this loss by moving the management to the hardware as in caches.

The latest research that attracted our interest and that is close to this topic was in [50]. The researchers introduce an innovative memory structure, stash, that exhibits the advantages of both caches and scratchpads but does not incur their drawbacks. Similar to a scratchpad, the stash can be directly addressed, it does not require the use of tags or TLB accesses, and also provides a compact memory space. The similarity with a cache is that their proposed memory structure, the stash, can be globally addressable with also the properties of visibility to the global runtime environment, supporting implicit data transfers and improving their reuse distances. In the aforementioned research paper that the stash delivers increased performance and with lower power consumption than a either using a cache or a scratchpad. In addition, the researcher claims that stash enables new use cases for heterogeneous systems. In the aforementioned research, 4 microbenchmarks are used in order to provide new use cases (e.g., reuse across Graphics Processing Unit (GPU) compute kernels). Moreover, the targeted programs compared to different configurations with scratchpads and caches. The results of the study show that stash can improve the execution time by an average of 27% and 13% respectively and energy by an average of 53% and 35%. For 7 current GPU applications, that do not include any compiler optimization support for the features stash component, in comparison with the execution times under scratchpad and cache configurations, it is proven that the stash reduces the execution cycles by 10% and 12% on average (max 22% and 31%) respectively, and energy by 16% and 32% on average (max 30% and 51%).

1.2.2 Implementations of High-Level Programming Languages

Our next step is to investigate the relation between the implementations of the runtime systems of High-Level programming languages and the memory subsystem. We focus mainly on Java, C++ that are widely used and on Haskell as it introduces a totally different programming concept. The details for Java and C++ will be presented in the relevant Chapters.

We took a close look at the implementation of Haskell. Haskell is a rather new functional programming language. The reason we focus on Haskell is because we are in search of more deterministic memory behavior. As stated in [28], Haskell has an interesting lesson to give about parallelism and determinism. Haskell replaces the use of concurrent threads and locks with the support of varieties of libraries that provide the programmer with a way to concisely produce high-level parallel programs. In addition, the results of the execution of Haskell programs are guaranteed to be deterministic. This determinism is not affected by the number of cores used nor the scheduling algorithm used in the runtime system. Although we came up with alternative ideas on how this totally different programming paradigm could enhance a processor's functionality, we continue only with the mainstream programming languages because we find Haskell's runtime system to be more demanding on the memory subsystem. The pressure originates from the relative big size of a closure, the basic memory structure used for the evaluation and application of the defined functions. In addition, despite the differences with object-oriented programming languages, there is a conceptual connection between the basic closure, the data structure used by Haskell's runtime system, and objects. Therefore some of the concepts we explore in this thesis could also be applied to Haskell.

During this literature research we observe two facts about modern high-level programming languages:

- Short-lived objects (or closures for Haskell) are a common case of modern programming languages
- Garbage collection is required for an Automated Memory Management system

It is a known fact that most object-oriented programs, during their lifetime, create many short-lived objects compared to the number of long-lived objects. As stated in [29], the high rate of object creation that exist in object-oriented computer systems results into high operational costs for the memory-management. Programs written in object-oriented programming languages exhibit the tendency to incur frequent procedure calls, which come at the cost of a number of pointer dereferences and updates, and they exhibit high requirements in memory space, typically producing dozens of memory space requests to compile and execute even a single line of code. Therefore, research in optimizing memory management has concentrated on new garbage collection algorithms. Nowadays, that good software techniques are available, future progress may have a higher dependance on hardware techniques.

1.2.3 High-Level/Tagged Computer Architectures

At this point we look into the implementation of High-Level Computer Architectures and the conclusions that were reached by the hardware designers of these processors. We investigate the design choices for High-Level Language Computer Architectures, focusing on their interactions with the memory subsystem. As referred in [30], High-Level Language Computers (HLLC) attracted interest in the architectural and programming research community, mostly between the 1970s and 1980s.

Most of the researchers at the time, justified their approach in High-Level Computer Architectures, pursuing one of the following goals as presented in [30]:

- Lessen the complexity of writing a compiler
- Exploit ways to produce more cost-effective systems
- Decrease the costs of the software development
- Bring closer programming and machine languages
- Decrease system software
- Efficiently execute HLL programs
- Enhance code compaction
- Ease debugging
- Exploit possibilities new architectures

The first tagged architectures in commercial use were Lisp Machines. As presented in [31], Lisp machines are computer systems that have special hardware support to efficiently execute Lisp programs. Lisp machines are general-purpose computing platforms. They are one of the first realizations of a high-level language computer architecture, and introduced the concept of commercially used single-user workstations. Although, they weren't produced in big volumes, Lisp machines were the first machines to support many now-commonly used technologies including effective garbage collection, laser printing, windowing systems, computer mice, high-resolution bit-mapped graphics, graphics rendering, and networking innovations like Chaosnet.

Around the 1960s and 1970s, the artificial intelligence (AI) community was developing software that demanded, what was considered, a tremendous amount of computing power and memory space to be executed. The requirement for more powerful computing platforms by the AI research community, was increased by the Lisp symbolic programming language. Moreover, at the same era designers were focusing also on hardware that was specialized to execute programming languages that were like assembly or Fortan. The high cost of developing these kinds of computer systems required the sharing of the platforms between many users. But in the same decades, the cost for designing and fabricating Integrated Circuits dropped, along with their decrease in size. In addition to the decreased costs of the improved IC technology, the memory requirements of artificial intelligence programs started exceeding the memory capabilities of the most common research computer, the DEC PDP-10. Considering those two facts hardware designers looked for new approaches to deal with this problem. One of these approaches was to design a computer system that would have as its main purpose to execute huge artificial intelligence software stacks and would be tailored to the semantics of the Lisp programming language. The main drawback of the implementation of this approach was that in order to keep the complexity of the development of the Operating System (OS) low, they would be owned by one user.

With the beginning of the "AI winter" along with increased usage of microcomputers that destroyed the business of the minicomputer and workstation manufacturers, Desktop Personal Computers became affordable. Moreover, these cheaper PCs were capable of executing Lisp software even faster that Lisp Machines. In addition, they did not require any special purpose hardware. This was a catastrophic fact for the Lisp Machine market. The hardware vendors that were specialized in this field viewed their business been eliminated. Other hardware oriented companies moved to developing software related solutions to survive extinction. Most of the companies that survived had a software

basis. Nowadays, Xerox and Symbolics are the only Lisp machine companies that still function and sell software related to Lisp machines.

Later on also the Java processors emerged. As Java programming language was becoming more widespread many researchers implemented the Java Virtual Machine in hardware. An example of such a computer architecture tailored to the Java language was the picoJava. As stated in [31], Sun Microsystems developed a microprocessor specification that was specialized in natively executing Java bytecode. The microprocesor requires the use of neither interpretation nor just-in-time compilation. The goal is to accelerate up to 20 times the execution of bytecode, in comparison to a common CPU running a Java Virtual Machine. PicoJava-based microprocessors were able to run C/C++ code as efficiently as a processor with RISC architecture. The aforementioned approach boosts the performance of Java's runtime system, and results into less memory usage but also delivering competitive performance for code written in other programming languages.

The end of High-Level language Computer Architecture came with the failure in terms of performance of the Intel's 432 processor. Generally, the main problem with these Architectural approaches was the fact that they were not cost effective. In [32], it is stated that this failure was also a consequence of the arrival of compilers with optimizations. Another factor, was the development of reduced instruction set computing (RISC) architecture and RISC-like CISC architectures. Moreover, just-in-time compilation for High-Level Languages(HLLs) emerged, that was another crucial software technique that improved the performance of common CPUs and thus did not justify any more research towards the field of HLLCs. Therefore, the designers of these complicated architectures found no benefit to compensate for the high cost of these exotic computer architectures. Moreover, by the end of their design the mainstream RISC processors had doubled their computational performance compared to the initial specifications that designers of Tagged or High-Level Language Computer Architecture had to meet. Especially for a Java processor the concept is doomed since Java's performance has come up to the point of being in the worst case two times slower or sometimes faster than a C based program. Thus, such a design would not be cost-effective.

1.3 Object Caches

During the literature research on High-Level Computer Architectures, we come across the interesting architectural concept of Object Caching. This approach involves the development of a special cache for manipulating or just caching the object structures.

Earlier research studies have shown that for programs written in Java a large fraction of memory references target the heap. In addition in [34], Kim and Hsu show that, on-average, 45 - 50% of the memory references requested from the SPECJvm98 benchmarks suite are addressing the heap memory. In [35], the researchers conducted experiments, again using different SPECjvm98 benchmarks, in order to measure the level of temporal locality that exists among the objects used by the targeted applications. They use the following definition for temporal locality: "The interval in terms of the number of accesses between two consecutive accesses to the same object". Indeed, Kim and Hsu observed a large amount of temporal locality in the access patterns of object references. They show that a huge portion of around 90 - 95% of all objects were accessed again after 100

accesses since they had been initially accessed. Moreover, a 40 - 80% of all references to objects involved the same object in the following memory reference. We consider this fact to be of great importance for the formulation of our concept of a Local Object Store, because it increases the probability for high hit rates in case a separate memory structure is used for objects. In the same research paper, the researchers investigate the lifetime of objects, that they define to be : "The time duration in terms of total number of (global) memory accesses between its first instantiation and its last access". The results show that most of the instantiated objects exhibit a short lifetime. The measurements reveal that using various benchmarks, 40 - 90% of the objects used exhibit a lifetime of less than 100 accesses. In addition, on the average 80% of all objects die in less than 1000 accesses. The Figures 1.1 and 1.2 depict the aforementioned findings in [35].

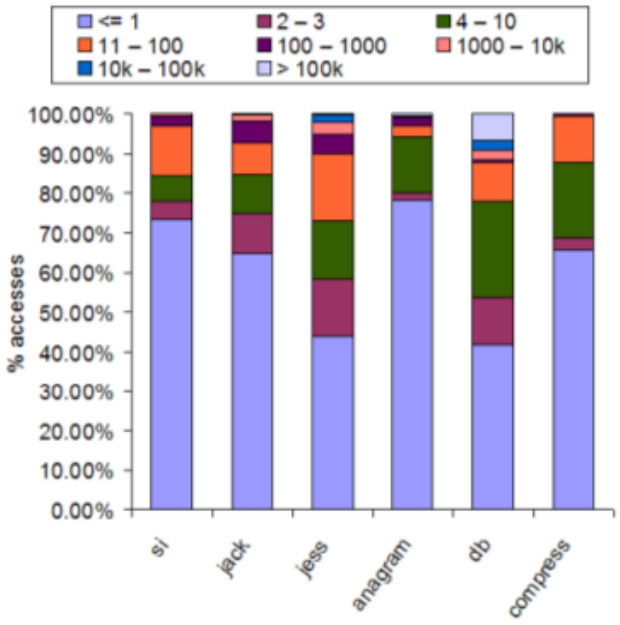

Figure 1.1: Temporal Locality of objects, source in [35]

In [36], the research focuses on developing a virtual object cache that decreases the time taken to perform an indirection while relocating objects efficiently. The indirection that is eliminated is linked to the utilization of handle representations to efficiently relocate the object, but introduces an additional cost of an extra indirection for every object read or write operation. To access objects inside the virtual address cache can be done using only the object reference along with the field offset. Thus, this approach removes the additional costs for the addition to calculate the offset and for the indirection. In addition, the operations of the proposed memory structure require the use of a hardware object table. The table keeps all handles and its' main purpose is to acquire the actual object location on a virtual address cache miss.

Close related to our research but an alternative approach is the investigation Morris Chang and Carl Lebsack did in [37]. The aforementioned research paper demonstrates

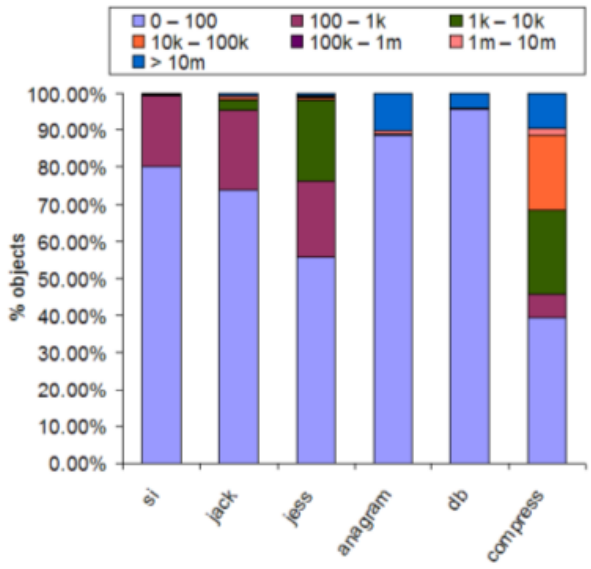

Figure 1.2: The Lifetime of objects across benchmarks, source in [35]

that using algorithms that perform generational garbage collection in a hybrid cache-scratchpad memory architecture can be beneficial compared to a common cache-only memory configuration. Their results show that the hybrid configuration exploits the locality of short-lived objects in Java and delivers a decrease of up to 20% for the main memory traffic. The most interesting claim in this study was that "Converting half of the cache to scratchpad can be more effective at reducing memory traffic than doubling or even quadrupling the size of the cache for several of the applications in SPECjvm98." This is another crucial factor that we keep in mind for our proposed memory unit.

Moreover, Morris Chang and Edward Gehringer in [29], propose the design of an application-specific coprocessor architecture that accelerates the allocation and relocation of objects for systems that are object-oriented. The proposed design utilizes a bit-vector for dynamically allocating and liberating memory space. When objects are created they are placed in a special cache that performs reference counting on them. The main advantage of this configuration is the fact that since mostly there are short-lived objects, they will be allocated and deallocated within the coprocessor resulting in a decrease of main memory traffic. The same researchers in [40] show the results for the proposed design. They simulated and proved that 50 - 70% of all objects that are instantiated, stay within the coprocessor's memory space during their whole lifetime. Thus, the outcome was the reduction of the references to main memory. Overall, they show a reduction of more than 60% for the main memory traffic and that the garbage-collection frequency drops.

Another interesting approach is presented in [50], the researchers introduce a memory unit that employs a garbage-collection technique. They utilize DRAM cells in order to enhance the performance of dynamic memory management operations. These operations involve allocating space for objects and keeping them reference-counted, as well

as clearing and compacting the memory space. In addition, another beneficial outcome is the predictability of the aforementioned functions. Space allocation is deterministic in terms of time, which means that it takes constant time to allocate an object within the module. Moreover, sweeping can be performed in parallel by various modules. The effect of the researchers' claims is that the sweeping operation requires constant time independent to size of the heap space. In contrast, the execution time of the software garbage collection techniques is proportional to the size of the heap memory. Another interesting design choice the researchers took was to limit the field for reference counting. This enabled the distribution of the required computations throughout the lifetime of the application. The main drawback of this approach is the increased power consumption. The requirements for power originate from the increased number of memory references and the complicated CPU architecture. Thus, the proposed module liberates the main CPU from the computations required to keep objects reference-counted. They use a quite cost efficient design requiring only 8,000 gates. Their research proves that using only 3 bits to count references of an object removes the need for calling the garbage collector for the targeted programs. Last but not least, an advantage of this configuration is the decrease in memory usage by 77 percent.

Object caches can help improve the Worst Case Execution Time of applications. In [39], Schoeberl proposed a separate object cache to allocate data on the heap. The researcher proposes a design with high associativity to follow symbolic object addresses in the static analysis. In addition, each cache line can keep only an object. Moreover, the design has a special operation to load atomic fields on a miss. The proposed cache can be analyzed statically which results in improved performance. In this paper the researchers show their design and implementation of the object cache under a single-core but also on a multicore version of the Java processor JOP.

The work in this master thesis tries to leverage the usage of a separate address space to manipulate objects as a scratchpad memory does. Compared to scratchpad memories, a common approach is that we map memory structures with temporal locality to our Local Object Store space. The main difference of our approach and the aforementioned implementations on Object Caching is the fact that we target a cooperation of the hardware and software in serving memory management techniques. Nowadays, due to the dropped cost of the technology we expect that the concept object caching reveals significant potential to enhance the memory management functionalities of a computer system. Although, the IC cost has dropped, we avoid extreme and costly hardware solution for generating object identifiers as the researchers exploring object caching in the earlier days of the concept did, for example, complete binary trees to generate an address. We base this decision on our intuition that these special techniques would result to higher operational costs in terms of power consumption. It is possible that these alternatives can be explored as future work, but it is our assumption that they will bring more drawbacks than advantages. In addition, we choose a basic structure for our initial study for a Local Object Store. There are many techniques that still can be incorporated to make the design more cost effective in terms of area. A common example would be to use a limited field for the tags, reference counters, of the objects. As mentioned in [38], the highest percentage of reference-counted objects will have up to 8 pointers referencing them. This gives us an opportunity to save die area by using

a limited tag field. Profiling of modern applications would better quantify the intuition that we have from our literature research.

1.4 Hypothesis

It is our assumption that dynamic heap allocation is the most important attribute of modern programming languages and presents a good opportunity to enhance the functionalities of a modern processor, while decreasing its operational costs in terms of memory access-based performance loss. We base our hypothesis on the literature research on the implementation of modern programming languages. Moreover, the advantages of garbage-collected programming languages and the research on High-Level/Tagged Computer Architectures suggests us to integrate the garbage collector's information in the hardware units of a processor that control the memory subsystem.

In this master thesis, we revisit the research questions that have been investigated during the early days of High-Level Language Computer Architectures. Since most of these approaches disappeared due to their costly designs, nowadays, that computer technology has entered the dark silicon era [41] and the IC costs have dropped there is renewed motivation for researchers to investigate more complex functionalities of the hardware. Most of the IC costs in modern processors are related to the memory subsystem. Processor designers dedicate a big fraction of their available area for the modules that serve memory operations. Thus, we base our hypothesis and add research value to the investigation on the concept of object caching and hardware garbage-collection techniques by assuming that the dropped cost in hardware could reveal these old concepts to be more advantageous for modern computer systems. In addition, our main hypothesis is that leveraging the "infant mortality" of objects, and using a separate memory address space for their birth and destruction can be more beneficial than placing them on caches which treat their address space as flat. We expect that the use of a Local Object Store will be beneficial in terms of reducing the memory traffic from and to the main memory. Moreover, we believe that since there are many short-lived objects in object-oriented applications, a garbage collected local store would be advantageous for most modern memory management systems. We form the hypothesis that a Local Object Store could provide the functionalities and the space for a crucial number of objects in real-life applications to die within the local address space without being purged back to main memory and therefore polluting the memory hierarchy.

We proceed in developing a Local Object Store that provides a partially automated memory management system on hardware. Moreover, we choose the simplest garbage collection technique, reference counting. We use fixed-size entries which is required for the objects to be reference counted. Thus, in order for our design to include garbage collection information requires a totally different design compared to caches.

We believe that our developed infrastructure can be the basis for future work on the cooperation of software and hardware in efficiently serving memory operations. For example, tag positions could be used as an alternative to software garbage collectors. In addition, as stated in [42], the use of identifier to address the memory space of objects, opens the way for more innovative virtual memory maganagement and garbage collection algorithms to be investigated. In addition, it is also claimed that the free choice on

identifier for the objects will enable designers to efficiently implement a direct software solution to translate virtual to real addresses as well as allocate objects. Thus, we believe that there are opportunities for more added research value in the future to the object caching architectural concept and proceed with an initial study as a proof of concept for the proposed Local Object Store.

1.5 Thesis Outline

The main contributions of this master thesis are:

- Modification of the HPROF profiling tool to log in-order the allocation and deallocation of objects. Moreover, we provide some additional modification that helps in tracking the changes of fields in the loaded classes, therefore tracking changes in the most important part of the garbage collector root set.
- The implementation of a "smart pointer" pointer library that can produce traces of reference-counted objects in C++
- The enhancement of the atomic memory trace Pin tool to log all function calls and exits
- An extension to instruction set architecture to support LOS functionality.
- A microarchitecture for a LOS
- A basic simulator for the comparison of cache-only and cache-LOS memory hierarchy

The remainder of the master thesis report is organized as follows. Chapter 2 presents an initial analysis in collecting statistical data from Java programs. This chapter also describes the reasons that force us to use C++ as our basic framework to collect traces. Chapter 3 discusses the implementation steps for a Smart pointer library and the modification of a pin tool to collect traces for the simulation of LOS. In addition, this chapter presents three microbenchmarks that will be used for our initial case study. Chapter 4 presents the microarchitecture of the LOS functional unit as well as the required modifications to the Instruction Set Architecture. Moreover, this chapter presents the developed software architecture along with the results of the simulator. Chapter 5, our final chapter, ends with conclusions and suggestions for future improvements.

Object Tracing with JVM Tool Interface

2

For the purpose of this master thesis research, we collect memory traces from benchmark applications. This information is required to simulate how the design of a Local Object Store functions and prove that we can benefit from locally operating on objects on the CPU. For a starting point, we choose to use the Java programming language, in order to collect real life application memory traces. Java is our primary target due to the fact that "everything" is an Object and it is dynamically allocated.

To collect the data of the memory traces, we need a detailed profiling tool. There are many profiling tools that can be used in order to collect memory traces and garbage collection information but the simulation of the Local Object Store requires detailed and in-order memory traces for the object allocation/deallocation and for memory accesses. To our knowledge there is neither a profiling tool available for the public that can collect this information in such a detailed manner nor a tool that can be tuned to provide these details. This is due to the fact that these profiling tools are intended to be used by programmers to fine-tune Java's Performance or to detect Memory leaks. For these two problems, developers require statistical data on the memory traces, which most of the times means aggregated results, i.e. the number of instances per object.

Thus, the first step to conduct our research is to develop such a tool and collect all the information we need at the level of detail appropriate to meet the requirements of the simulation of the Local Object Store. An alternative path that possibly can be followed is to collect the memory traces of the target benchmarks by altering the JVM implementation. This approach would definitely serve our demands but is abandoned due to the uncertainty in terms of time and effort required to alter the open-source implementation of the Java Virtual Machine, the OpenJDK.

To sum up, in this section the background of the Java Memory Management, especially the parts relevant to operations that regard object handling will be presented. Then we describe the framework used, the JVM Tool Interface. Afterwards, we present the designed software tools that we create. In the end, this section will illustrate the results of the memory traces and more information on the outcome of collecting all these data, as well as the reasons, we move our research into developing and utilizing a C++ smart pointer library for the required collecting information on Reference Counting.

2.1 Background on Java Memory Management

In this section, we describe the Automatic Memory Management of the implementation of the Java Virtual Machine before diving into the details of the implementation of the profiling tool.

In the Java programming language programmers tend to refer to the fact that everything is an object, and has super class the "java.lang.Object", which indeed has no

other super class, and they stay on the heap. This is almost true except for parameters and local variables of a method which are live on the stack and are allocated during the invocation of the method call and deallocated when the method returns. In addition to the local variables and the parameters, there are more exceptions including primitives and references. Finally, in order to be precise fields(not the data but the structure of the object), methods, generic classes and blocks of code are also excluded. This statement refers to the fact that most of the data live in the heap under the Memory Management of objects. In Java, most of the times, one cannot express any solution without using objects and generally using Java and being modular in programming means the creation of a lot of objects.

The objects live and die within the heap. As presented in [44], the JVM keeps two separate memory spaces, the Java and the system heap. Their purposes as well the mechanisms used for their maintenance are totally different. All objects that are instantiated during the lifetime of a Java program are located in the Java heap, commonly referred as "heap". This memory space is allocated with the use of "mmap". The JVM uses "shmat" in case that the use of large pages is demanded and supported. In addition, the JVM preallocates, during its startup, a contiguous area which represents the maximum space that can be used for the Java heap. Moreover, the aforementioned allocation with the use of heap expansion moves the artificial heap size boundary towards the real heap size boundary. This occurs even under the selection of the smallest space choice for the heap. The Garbage Collector is responsible for maintaining the heap. In addition, the commonly used heap settings in the command line refer to the size of the Java heap. Memory space for the native heap is allocated with the use of the malloc and free operations, the common system calls, and serves the purposes of the hidden implementation of some Java objects; a few examples are:

- Motif objects that are used by AWT and Swing
- Special Buffers used in compression algorithms, which are the memory structures that the Java Class Libraries use to access compressed data like .zip or .jar files.
- Allocating memory by Java Native Interface(JNI) code with Malloc
- Ready for execution code produced by the Just In Time (JIT) Compiler
- Threads to map to Java threads

The following figure depicts the location of the heap in the memory layout of the JVM.

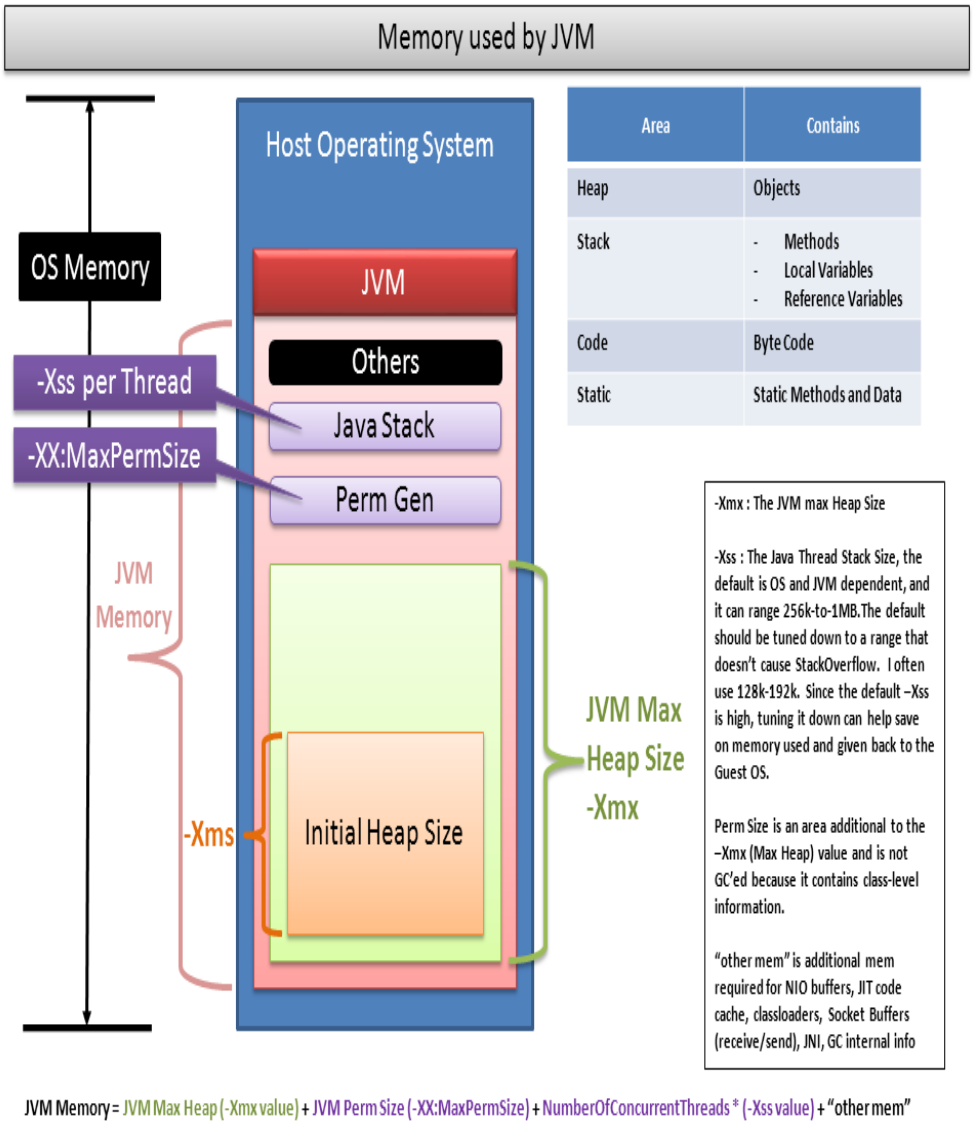

Figure 2.1: JVM's Memory Layout, source in [45]

In addition, the heap is separated into different memory spaces that the garbage collector is in control. This is mostly because of the use of generational garbage collectors. Based on their lifetime, the garbage collection algorithm moves the objects between the different memories spaces. The following figure from [46] shows the different generations and memory spaces within the heap of the JVM implementation.

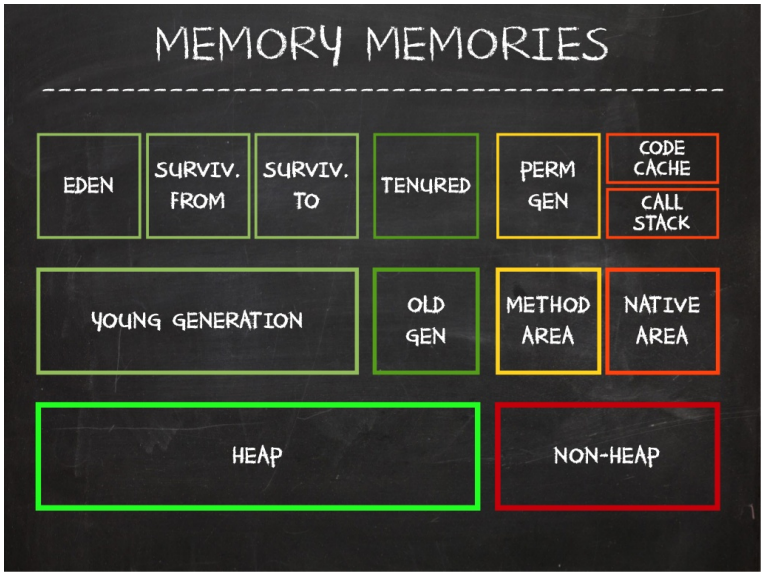

Figure 2.2: Memory areas within and outside the Heap, source in [46]

Java Objects are in the central point of the Automatic Memory Management of the JVM's implementation. Allocation of new objects can be done using the "new" operator. The new operation instantiates a class by allocating memory for the class in the heap, calling the constructor in case it's not an object and returning a reference to that instantiated class(object). Specifically, from the section on new instance creation of the JVM Specification, it can be understood that this is the sequence when a new object is created, as long as no exceptions occur, as presented in [2]:

"**A)** Enough Memory space is allocated for all the instance variables declared in the class type and all the instance variables declared each superclass of the class type, including any hidden instance variable. If there is not enough space an OutOfMemoryException is thrown.

B) The instance variables are initialized to their default values.

C) The indicated constructor is processed in the following order:

1)Assign the arguments for the constructor to newly created parameter variables for *this* constructor invocation.

2)If *this* constructor begins with an explicit constructor invocation of another constructor in the same class (using *this*), then evaluate the arguments and process that constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason; otherwise, continue with step 5.

3)This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using this). If this constructor is for a class other than Object, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using super). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue with step 4.

4)Execute the instance initializers and instance variable initializers for this class, assigning

the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception. Otherwise, continue with step 5.

5)Execute the rest of the body of this constructor. If that execution completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.

D) Return the reference to allocated space in the Heap Memory. "

On the other hand, deallocation is not explicit because Automatic Memory Management is used by the JVM implementation. Objects instances live inside the Heap. When the JVM starts executing an application the Heap increases in size as new objects are created by the program. When the Heap becomes full it calls the garbage collector pausing the application. The garbage collector will free up space from the Heap by detecting objects that are considered to be garbage. Objects are considered to be no longer in use by their reachability. An object is regarded to be unreachable if it cannot be accessed by chasing pointers of live data. At any given point of a program's runtime, liveness is considered to be the reachability of an object from the root set. The root set includes the global variable, for example the static fields of a class or the fields of the main thread, and any local variable of all live method activations(i.e. the stack). To be absolutely precise about what consists the Garbage Collection Roots we can refer to IBM's knowledge center at [3]:

"A garbage collection root is an object that is accessible from outside the heap. Memory Analyzer categorizes garbage collection roots according to the following list:

System class
A class that was loaded by the bootstrap loader, or the system class loader. For example, this category includes all classes in the rt.jar file (part of the Java runtime environment), such as those in the java.util.* package.

JNI local
A local variable in native code, for example user-defined JNI code or JVM internal code.

JNI global
A global variable in native code, for example user-defined JNI code or JVM internal code.

Thread block
An object that was referenced from an active thread block.

Thread
A running thread.

Busy monitor
Everything that called the wait() or notify() methods, or that is synchronized, for example by calling the synchronized(Object) method or by entering a synchronized method. If the method was static, the root is a class, otherwise it is an object.

Java local
A local variable. For example, input parameters, or locally created objects of methods that

are still in the stack of a thread.

Native stack
Input or output parameters in native code, for example user-defined JNI code or JVM internal code. Many methods have native parts, and the objects that are handled as method parameters become garbage collection roots. For example, parameters used for file, network, Input/Output(I/O), or reflection operations.

Finalizer
An object that is in a queue, waiting for a finalizer to run.

Unfinalized
An object that has a finalize method, but was not finalized, and is not yet on the finalizer queue.

Unreachable
An object that is unreachable from any other root, but was marked as a root by Memory Analyzer so that the object can be included in an analysis. ”

Indeed, a safe policy is to delete any unreachable data as it will never be used by the program again. The side-effect of this policy is to retain every reachable object as it is possible that it will be accessed in the future, which can cause Memory Leaks.

After the garbage collection finishes clearing the Heap Memory space the application can continue running. This fact is called stop-the-world and occasionally is one the reasons which hinders the performance of an application. It has to be noted that there is more memory space used by the Java Virtual Machine(JVM) except for the Heap Memory for methods, native handles and thread stacks and for JVM internal data structures are allocated separately from the heap.

There are a number of different garbage collection algorithms used for their advantages and disadvantages. Some of the most common ones and their attributes are the following as presented in [4],[5],[6] and [7]:

1) Reference Counting

Pros:

- Incremental Technique
- Ensures that every objects will be deallocated at the moment its last reference is eliminated
- It does not have a huge impact on the CPU cache and the virtual memory operation, mainly because it requires the access to information in the memory that is either local in the caches, inside objects to be deallocated, or directly pointed by these objects
- Small overhead per memory write
- Does not pause the application and has clearly defined lifetime for every object. This is a big advantage for real-time applications and systems that have limited memory space because it maintaines repsonsiveness
- Simplest forms of Memory Management

Cons:

- Frequent updates
- Space overhead for reference counting information to be maintained
- Reference Cycles can't be handled
- Cascading decrements can be expensive
- Free space is never compacted
- Because free space is not compacted, a "free space" data structure must be maintained which increases allocation and deallocation costs
- Updating reference counts in a multi-threaded application is extra expensive

Reference Counting requires additional mechanisms to ameliorate these disadvantages. For example, reference cycles can be avoided by using strong and weak references,thus a Parent to Child object reference would be a strong one and Child to Parent a weak reference.

Tracing Garbage Collection Techniques

The next big class of algorithms that perform garbage collection, involve the tracing of object references from the root set. Using pointer tracing ensures the reachability property for all the live objects and points out the dead objects. There are a number of different techniques and combination of them that produce hybrid algorithms. The most common ones are the following:

2) Classical Mark and Sweep

Pros:

- Handles reference cylces
- Memory writes/dropped aliases have no cost
- Storage management concurrency bottlenecks can be avoided (apart from GC)

Cons:

- Fragmentation
- Stop-the-world garbage collection
- Overhead is propotional to the heap size, because sweep phase requires to traverse the whole heap
- Free space is never compacted
- Because free space is not compacted, a "free space" data structure must be maintained which increases allocation and deallocation costs

3) Mark-Sweep-Compact

Pros:

- No pointer write overhead
- Cyclic data is collected
- Storage management concurrency bottlenecks can be easily avoided (apart from GC)
- Free space is compacted, so allocation is cheap

Cons:

- Stop-the-world garbage collection
- The compact phase is rather expensive

4) Copying Garbage Collection

Pros:

- Only deals with live data
- No Fragmentation due to automatic compaction, which will probably increase the locality

Cons:

- Requires double the memory space compared to Mark and Sweep
- Stop the world Garbage Collection

5)Generational Garbage Collection described in [8]

Pros:

- Takes advantage of the infant mortality or generational hypothesis, which means that there is a higher probability for the younger objects to become unreachable compared to the older ones
- Older Generations collected less often
- Cyclic data is collected
- Storage management concurrency bottlenecks can be easily avoided (apart from GC)
- Usually does garbage collection on just part of the heap with lots of garbage, and hence GC overheads are less on-average
- Smaller GC pauses (most of the time)
- Free space is compacted, so allocation is cheap
- Compaction comes more cheaply than with a sliding compacter

Cons:

- It is based on heuristic methods. This means that when generational garbage collection techniques are employed, it is not guaranteed that all unreachable objects will be reclaimed every cycle. There, occasionally it is required to call a full mark and sweep or copying garbage collection in order to to reclaim all free space.
- Space overhead due to the need to reserve an extra object space for the collector
- Pointer write overhead (to record when a "new generation" pointer is written into an "old generation" object)
- Stop the world Garbage Collection

It is a fact that most modern programming language implementations that use automatic memory management do have a hybrid generational garbage collector which implements different algorithms for the different generations. One problem with "Stop-the-world Garbage Collection" is that it has to halt the execution of the application to run the collector in order to guarantee that no new objects will be allocated or objects suddenly becoming unreachable while the garbage collector is running. In addition, the main disadvantage of the "stop-the-world" is the fact it has a big performance impact for parallel programs. For this reason, concurrent garbage collectors were developed. There are also the Parallel garbage collectors but they still halt the execution of the process. Their attributes, as stated in [11] are described in the following text.

6) The Parallel/Throughput collector

The biggest advantage of this algorithms is the use of multiple thread to do pointer chasing and in addition to that compact the heap space. The Parallel Collector is a "stop-the-world" technique, thus stopping all the threads of an application. Therefore, it best suits applications that tolerate pauses and have as optimization goal to lower

CPU overhead caused by the collector.

7) The Concurrent-Mark-Sweep Collector

This technique also uses multiple threads to traverse the pointers through the heap for dead objects that be collected. There are two cases that the Concurrent-Mark-Sweep (CMS) collector can enter the "stop-the-world" mode and pause the application's execution:

a) When initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables)

b) When the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked.

A common drawback that may be encountered while using the CMS are promotion failures. These race conditions occur between collecting the young and the old generation. In case the algorithm has to promote young objects to the old generation, but hadn't had enough time to clear space for it, then it will have to make space first which will result in a full "stop-the-world" collection.

Another downside of this collector, compared to the Parallel one, is that it increases the overhead for the garbage collection in order to provide the application with higher levels of continuous throughput, by using multiple threads to perform scanning and collection. For most long-running server applications which are adverse to application freezes, that is usually a good trade off to make. This collector should be preferred in case the programmer has the availability to allocate more CPU resources. Last but not least, the max heap size should not exceed 4GB or else in that case it is probably better to use the Garbage First Collector.

8) The Garbage First Collector

The Garbage First (G1) Collector utilizes multiple background threads to scan through the heap that it divides into regions, starting with sizes of 1MB to 32MB depending on the size of heap the application uses. This technique is geared towards scanning those regions that contain most of the garbage objects first. This strategy avoids the chance of the heap being depleted before background threads have finished scanning for unused objects, in which the collector will have to stop the application which results in a "stop-the-world" collection. Another advantage of the G1 is also that it compacts the heap on the go, something the CMS collector only does during STW collections.

We need to develop a framework upon which we can incrementally exploit the trade-offs on how all these alternatives could help us into the cooperation of hardware and software for memory management techniques. Thus, we do believe that referencing counting is our best choice. Except for hardware simplicity, reference counting can minimize die area by using limited fields for the tags.

2.2 Description of Java Virtual Machine Tool Interface

For the purpose of simulating the functionality of the Local Object Store, we require memory traces from real-life applications written in Java programming language. In order to collect this information from the runtime system of the JVM, we utilize a programming interface that enables the programmer to develop profiling and monitoring tools for the applications, the Java Virtual Machine Tool Interface (JVMTI). As stated in [9], JVMTI provides a VM interface for all kind of tools that require privileged access to the Virtual Machine state, some examples are:

- Profiling
- Debugging
- Monitoring
- Thread analysis
- Coverage analysis tools

JVM TI has a two-way interface to perform the communication between the agent (profiling tool) and the VM running an application. The JVMTI informs the agent, its' client, about interesting occurrences through events. JVMTI does control and check the state of the application through various functions responding to events but also independent of them.

In addition, all the agents are executed in only one process. While testing the targeted Java program, agents gather information directly from the virtual machine. A native interface is used for the underlying communication mechanism between the JVM and the agents. Normally, the code size of the agents is quite reasonable. Different processes implement the bodies of the functions related to the tools. The aforementioned executions of the processes do not interfere with the normal control flow of the Java program run by the JVM.

In simple words, the agent has to export specific functions to provide the callbacks for the JVMTI. For example, in order to start the agent the library must export the following function callback:

JNIEXPORT jint JNICALL
*Agent_OnLoad(JavaVM *vm, char *options, void *reserved)*

The aforementioned method is called when the VM is being initialized. This convention is required to meet some fundamental requirements:

- System properties must be first set in order to be used
- There is availability of all the capabilities
- The VM has not run any bytecode instruction
- The VM has not perform any class loads
- The VM has not allocated any object

There is the possibility of simultaneously calling many JVM TI agents in a JVM. The JVM ensures that separate JVM TI environments are used by every agent.
As stated in [9], the JVM TI environment state incorporates:

- The event callbacks
- The set of the enabled events
- The capabilities
- Hooks for memory allocation and deallocation

Another fact about the JVM TI is that not all events can by captured with the JVM TI callbacks. For example, object allocation events or full speed enter and exit events are not handled through callbacks. Instead, the interface supports bytecode instrumentation in order to insert callbacks before the uncovered events.

As stated in [9], there are three ways to instrument a code file:

- Static Instrumentation:
  The targeted class file is being duplicated. Precisely, the VM creates a separate folder to keep all the files altered by the instrumentation. The main drawback is that the agent has no knowledge for where the loaded class file originates.
- Load-Time Instrumentation:
  While the JVM loads a class file, the file as raw memory is dispatched to the agent. The agent will proceed with the instrumentation receiving a ClassFileLoadHook event. The event of loading of a class invokes the callback within the agent. This method is an efficient way to support complete access to one-pass instrumentation.
- Dynamic Instrumentation:
  This technique refers to the capability of altering a class file after it has been loaded on the JVM. In addition, it is also possible to perform dynamic instrumentation during the execution of a program. Another interesting feature is that the files can be altered several times and can be also turned back to their original state.

Initially, we limit the scope to only Load-Time Instrumentation. We proceed with understanding the library to collect memory traces and perform statistical analysis on the objects' properties for modern real-life applications.

2.3 HPROF and Our Modifications

In order to understand and quickly extract the traces through the JVM TI we use and modify the HPROF profiling tool. HPROF is an open-source tool widely used by JAVA programmers to perform tuning of the runtime behavior of the their Java applications. The source code guided us into understanding how to setup the capabilities. Bytecode instrumentation is also performed by calling the JAVACRW demo with the right set of arguments pointing to the class to be instrumented. Through the same way HPROF performs the bytecode instrumentation of targeted Java applications.

Our modifications are simple since the HPROF already collects the object allocation traces and aggregates the info. The basic modification to the functionality of HPROF that we do is to collect all the object allocations in a file stream. In addition, we do collect information about each object during runtime. The required changes are shown in the following C++ source code:

```
/* Record an allocation. Could be jobject, jclass,
   jarray or primitive type. */
static void
any_allocation(JNIEnv *env, SerialNumber thread_serial_num,
               TraceIndex trace_index, jobject object)
{
     /* HPROFs Code*/
     /* Inserted code */
     uid=(jlong)getTag(object);
     char *declaringClassName;
     getClassSignature( klass, &declaringClassName, NULL);
     fprintf(stderr,"B!%X!%X!%X!%s\n",(long)uid,(int)size
                    ,thread_serial_num,declaringClassName);

}
```

This is the main callback from the bytecode instrumentation event that will be called in case of any allocation. At this point we need to explain what is the tag of an object. A tag is a number correlated to an object. Agents can make explicit declaration of the values for the object tags. To define tags the *SetTag* procedure can be called. Another way to define the contents of the tags is through the use of special callback functions like the *jvmtiHeapIterationCallback*. The tags are placed in the agent's environment. Thus, the tags set by an agent cannot be viewed by another. Moreover, tags can be used just for object marking purposes as well as for pointing to more interesting information and their type is *jlong*. The initial value of the field is always zero. Thus, a zero tag denotes that the object is untagged. For the purpose of our profiling tool we use this tag as the Universal Identifier (UID) of the object. The aforementioned inserted code, retrieves:

- The tag(UID)
- The type of the structure to be allocated
- The size of the allocation
- The serial number of the thread that the allocation was requested

Moreover, another three print commands were inserted for the following events:

- Thread Start. The name of the thread and its serial number are printed.
- Carbage Collection start.
- Carbage collection. Prints also the time since GC started.
- (JVM TI callback) Object Freed. Prints its tag.

Although, we do not reach the level of detail in the acquired information for a simulation of a Local Object Store, our modifications allows us to collect all the information

for the statistical analysis of the following section.

2.4 Information on object size

We use our modified version of the HPROF utility to collect object allocations/deallocation traces and information on the size of these allocations. We use the traces to perform statistical analysis on the percentage of object that could be instantly allocated within a Local Object Store. We develop a small program to analyze the traces. The program aggregates the number of objects that have a size of equal or less than a fixed size LOS entry, which we define to be 64 bytes for an initial case study. For the purpose of this analysis we use DaCapo Benchmark suite. As described in [47], DaCapo is mainly used by the programming language, memory management and computer architecture research communities. All targeted applications represent real world software stacks and they are open source. The memory footprints of these programs are quite demanding and pressing the memory submodule. The DaCapo-9.12-bach benchmark suite, was released in 2009, consists of the following benchmarks as presented in [48]:

avrora
is a simulation of a number of different applications executed on a grid of AVR microcontrollers

batik
depends on the unit tests in Apache Batik to produce various Scalable Vector Graphics (SVG) images

eclipse
is a set of test cases for the (non-gui) jdt performance of the Eclipse IDE

fop
receives files of XSL-FO type, to parse them and format them, producing a PDF file.

h2
is an in-memory benchmark similar to JDBCbench, performing various transactions to test the model used in a banking application

jython
translates a pybench to a Python benchmark

luindex
Utilizes lucene to index books of Shakespeare and the King James Bible

lusearch
Utilizes lucene to perform a text search for keywords over the books of Shakespeare and the King James Bible

pmd
inspects a number of Java programs for various problems in their source codes

sunflow

renders a collections of images with ray tracing

tomcat
sends queries to a Tomcat server acquiring and validating the produced webpages

tradebeans
executes a benchmark, called daytrader, using a Jave Beans to a GERONIMO backend. Moreover, for the hidden database, an in memory h2 is used.

tradesoap
executes a benchmark, called daytrader, using SOAP to a GERONIMO backend. Moreover, for the hidden database, an in memory h2 is used.

xalan
performs the tranformation of XML to HTML documents

The statistical results can be seen on the following tables:

Bench\Size	16	24	32	40
avrora	8611	572839	1272312	7683
batik	79135	243534	333102	88849
eclipse	48834	715345	275393	155660
fop	85329	1394336	481466	239934
h2	2773358	1846890	691925	1895643
jython	61073	1108979	220649	1899591
luindex	6499	131368	38456	76021
lusearch	216963	1314191	397172	522880
pmd	130326	737568	311579	100300
sunflow	6100	551391	83206	3395
tomcat	47946	505079	331133	110265
tradebeans	230491	724820	311397	199949
tradesoap	229893	706012	306932	198546
xalan	32962	1135494	404348	156908

Table 2.1: Results of the analysis of The Dacapo Benchmark

Bench\Size	48	56	64	< threshold	total
avrora	19571	5029	4504	1890549	1952471
batik	108726	56572	21394	931312	1198335
eclipse	148832	82356	63722	1490142	1952544
fop	261572	159730	41837	2664204	2913878
h2	109774	102521	134779	7554890	7930505
jython	409985	67737	92684	3860698	4079052
luindex	27044	12805	7047	299240	364323
lusearch	333930	127297	113650	3026083	3695142
pmd	228023	65982	208503	1782281	2007407
sunflow	1385056	44007	1694	2074849	2094740
tomcat	155054	447871	63467	1660815	2095207
tradebeans	184853	87828	70932	1810270	2707730
tradesoap	181899	83444	69549	1776275	2631226
xalan	1214837	28127	32168	3004844	3238408

Table 2.2: Results of the analysis of The Dacapo Benchmark

From the results, we observe that most objects could fit within the entries of a LOS. We depict this outcome on the following table and graph:

	In LOS			In LOS
avrora	96.83		lusearch	81.89
batik	77.72		pmd	88.79
eclipse	76.32		sunflow	99.05
fop	91.43		tomcat	79.27
h2	95.26		tradebeans	66.86
jython	94.65		tradesoap	67.51
luindex	82.14		xalan	92.79

Table 2.3: Table on percentages of Objects that can fit with a LOS (1)

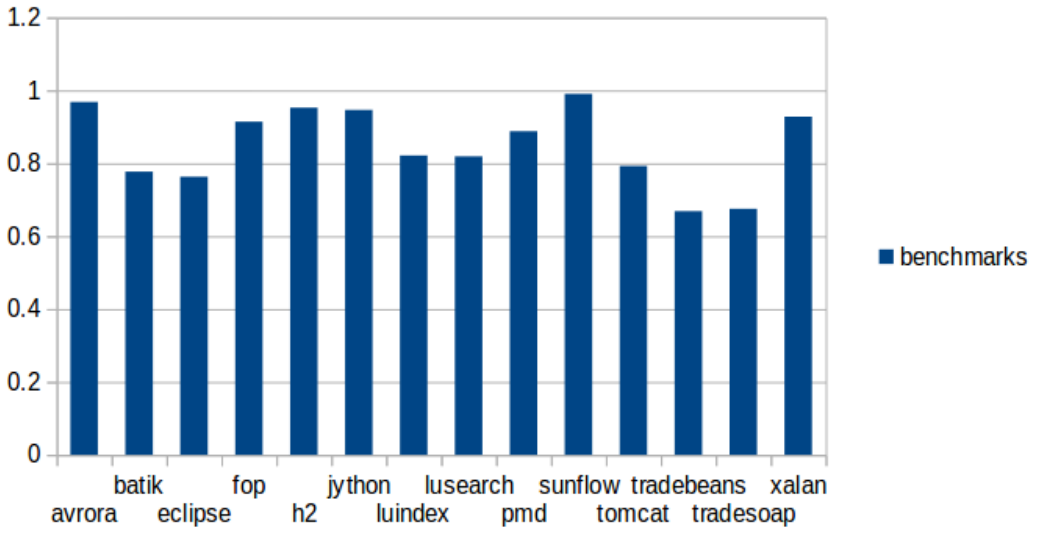

Figure 2.3: Graph on percentages of Objects that can fit with a LOS (2)

Thus, in this section we validate the findings of our literature research. Future work involves validating also the findings on object lifetime.

2.5 Proposed changes to add tracing of references

First of all, the main issue is that we capture events from objects allocated in the heap and there is a large amount of objects that are allocated and die within the stack. Moreover, to do reference counting it is required to follow the changes of the root set of the garbage collection. Therefore, in order to include all objects and perform reference counting on them, our agent has to be modified to check all the stack traces for objects and pointers allocated and determine their lifetime in terms of how many memory allocations happened between the time when the object was created and when it was destroyed. An alternative choice could be the modification of an open source Java Virtual Machine. Due to time limitations for completing this master thesis, none of these approaches are investigated. We proceed with using the C++ programming language to define all required software parts for the simulator and tracing tool, thus we come up with a simpler framework to formulate our proof of concept for the benefits of a Local Object Store.

C++ based object tracing framework

3

The difficulties of following the full root set of a Java application lead us to search for alternative ways to collect traces on reference-counted objects. At this stage, we change course and switch from a programming language with automatic memory management to the programming model of C++, that is based on manual or library-based memory management. Moreover, we modify a tracing tool based on Intel's PIN library to collect traces for the simulator of a memory architecture that includes a Local Object Store.

3.1 Memory management in C++

C++ was created by Bjarne Stroustrup as an evolutionary extension to the C programming language. Thus, at its heart, it is a low-level programming language. Moreover, another factor that defines C++ to be low-level is the support for abstractions that have zero overhead. This is a result of incorporating the C subset in the language. Stroustrup referred to C++ as "a light-weight abstraction programming language for building and using efficient and elegant abstractions"[12]. Being an extension to C, C++ supports four types of memory management as presented in [13]:

a) Static storage duration objects

These types of objects are created before entering the main() function and are destroyed in reverse order of creation after main() exits. The static objects have lifetime spanning the execution of the program. Initializing static objects refers to two stages. The first step is "static initialization" and refers to initialization with zeros of the memory space of all objects and after which comes the initialization of the constant expressions. This step can be performed at compile time. The next phase is dynamic initialization for objects that require the call of a constructor or a function call (unless the function is marked with constexpr). In general, static variables and objects are global declarations.

b) Thread storage duration objects

These objects are similar to are similar to the previous type, but their lifespan is the thread's lifetime and not the application's execution time.

c) Automatic storage duration objects

This category includes mostly local declarations that exist within a scope, block of code or a function but also temporary variables. Their lifetime is limited to their scope.

These local and temporary variables and objects are created and possibly initialized at the site of their declarations but destroyed in the reverse order of creation as soon as the execution of their scope is ended. This discpline is termed as RAII, "Resource Acquisition Is Initialization". In RAII, C++ destructors for local variables are called at the end of the object lifetime.

d) Dynamic storage duration objects

These objects are created with the *new* operator call and destroyed with an explicit call to the delete operator.

The heap is located in the main memory of a C++ program. In [48], the author depicts the memory layout of a C++ process on a Linux based operating system, we can see the location of the heap in figure 3.1.

C++ is known to be one of the first widely used Object-Oriented Programming languages. Thus, in addition to the C features, C++ includes some basic OOP features:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

C++ also supports deterministic destructors. This is one of the most characteristic attributes of C++ Classes compared to the implementation of Classes in other object-oriented programming languages. This property enables C++ programmers to use the Resource Acquisition Is Initialization (RAII) Concept.

As mentioned in [13], when RAII is used, keeping an object/resource is related to its lifetime. In the same context, acquisition means object/resource allocation and its done by a call to the constructor of the class. On the other hand, release is translated to deallocating the resources by a call to the destructor of the class. Resource leaks can be avoided if all objects are terminated in the appropriate way. The object/resource lifetime might be linked to the scope of its acquisition and release, but it is commonly used for the implementation of automatic variables.

An advantage of selecting RAII out of the various resource management methods is the support for encapsulation as the underlying logic that performs resource management, and it is not required to be determined for every call site but only once inside the Class. In addition, exception safety can be provided for resources that are instantiated within the stack. These type of resources have a lifespan related to the scope of the function that allocated them. The exception safety can be supported by linking the lifetime of these resources to another stack variable. The programming convention behind exception safety is that the destructors of the objects will be the last call when exiting a scope. Last but not least, the constructor and the destructor of a class have adjacent definitions. Thus, an application can locate during runtime the definitions for the constructor and the destructor of a class.

Thus, it is essential that resource management is linked to the lifetime of appropriate objects to support automatic allocation and reclamation. Allocation happens with a call

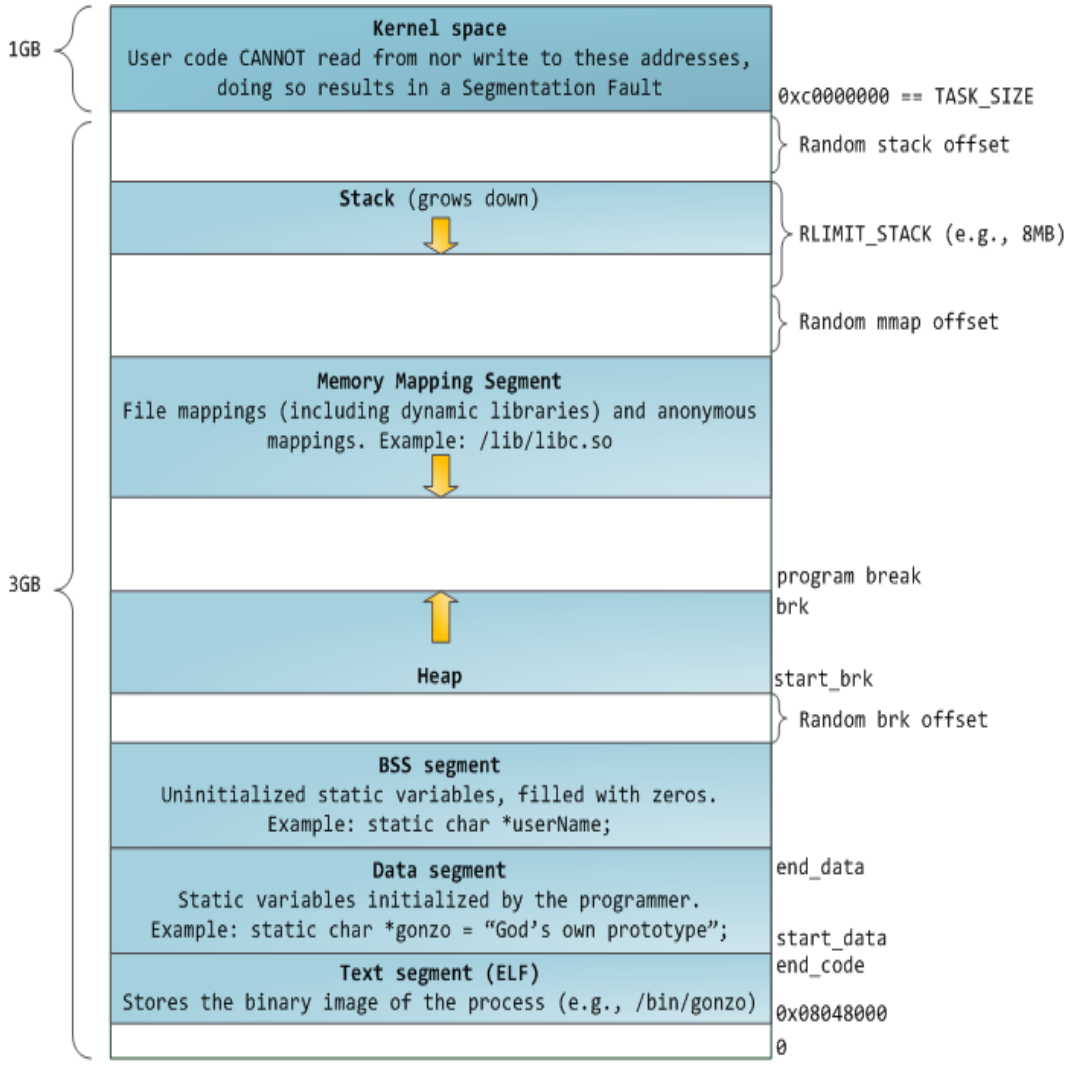

Figure 3.1: Memory layout of a C++ process in a Linux based OS, source in [48]

to the constructor of the class that initializes the object before it can be used. Object deallocation occurs by calling the destructor. A call to the destructor will also happen in case of any error.

Stroustrup comparing Java's *finally* construct with RAII stated in [15] that "In realistic systems, there are far more resource acquisitions than kinds of resources, so the *resource acquisition is initialization* technique leads to less code than use of a *finally* construct."

RAII has the ability to handle objects that are solely located in the stack. It requires the implicit or explicit destruction of objects that reside in the Heap space. Thus, it is the programmer's responsibility to call the destructors of all heap-allocated objects or the developers can use library-based automated management for these objects such as "smart pointers". In addition, weak-pointers can be used to eliminate cycles of references.

The definition of "smart pointers" as described in [16] is: "an abstract data type that simulates a pointer while providing additional features, such as automatic memory management or bound checking." The aforementioned extra features are related to efficiently catching exceptions and eliminating various "bugs". These type of pointers are tracking the memory space which they reference. In addition, there are cases that "smart pointers" can be used for handling different type of resources, such a file handle or a network connection.

Generally, pointers can introduce many "bugs" during the development of an application, due to the fact that it is quite easy for a programmer to misuse them. On the contrast, using "smart pointers" eliminates these sources of problems because the destruction of the resources is automatic. "Smart pointers" prevent Memory leaks. Thus, every object destruction is guaranteed to happen if and only if when the last owner of that object is eliminated. Another advantage is that, "smart pointers" remove dangling pointers by waiting until the object is left unused.

There are various types of "smart pointers". They can operate either by using reference-counted objects or by maintaining for every object a single pointer as an owner. The main two categories of "smart pointers", as implemented in the C++11 release, are the *unique_ptr* and the *shared_ptr*.

Unique pointers contain a raw pointer that is "owned" by the *unique_ptr*. Moreover, the data referenced by a unique_ptr cannnot be copied. This is in contrast to the normal assignment operation. In case the data are required to change possession the *move* function of the C++ standard library (*std*) can be utilized. Thus, there is no declaration for the copy and assignment operators. On the other hand, a *shared_ptr* also contains a raw pointer that keeps ownership using reference counting. In addition, shared pointers maintain all copies of the *shared_ptr*. This is required to ensure that the object referenced will be destroyed after all copies of the shared pointer have been destroyed. Another type of "smart pointers" are the aforementioned weak_ptr, which is used for eliminating cyclically referenced objects. These are special-case type of pointers that can be used in conjunction with shared_ptr as presented in [17].

3.2 Describe our framework, SP library

In the previous section, we conclude by referring to "smart pointers" and their use to handle heap-allocated objects that do not have a specific ownership. Considering the scenario of collecting reference counting traces, the "smart pointer" library seems to be an ideal alternative to post-processing memory traces from the JVMTI. Moreover, writing a basic "smart pointer" library, instead of a JVMTI agent, requires less time and effort.

Thus, we create a basic "smart pointer" library named SP to serve the needs of the Local Object Store simulation. The main feature of C++ that permits the main functionalities of the SP library is the ability to overload operators. This programming technique is also used by other libraries that have "smart pointer" implementations, for example boost library. Moreover, we base the development of our SP library on the guidance of [49].

Our SP library implementation of "smart pointers", except for dealing with all the

appropriate functionality of reference counting on objects, also adds the feature of collecting traces for interesting actions, like object allocations and deallocations. The first step that we take to create the SP library is to define a class that stores the reference counting variables and methods:

```
class RC
{
   public:
    int count;        //Variable holding the count number
    void AddRef()     //Method for adding a reference
    {
        count++;
    }

    int Release()     //Method to decrement the reference counter
    {
        return --count;
    }
};
```

Our next step is to build the main SP library that is based on templates, but before we proceed, we make a programming convention to be able to nullify a SP pointer. To do so, we define a base class SP_null upon which our SP library will extend and a global declaration of a SP_null class named MyNull. Thus, the basic declarations for both classes and for MyNull will be:

```
class SP_null{};
SP_null MyNull;
template < typename T > class SP   : public SP_null
{
private:
    T*     pData;        // pointer to data
public:
    RC* reference; // Reference count

};
```

The declaration of SP_null class is empty because it is only used to distinguish the type of the SP pointer. In the case that the type of a SP pointer declaration is SP_null, it is denoted that the pointer is null. MyNull is a public declaration that is used as the return value when calling the function (assignment operator) that nulls a SP pointer. In addition, the SP class extends SP_null for type equality issues. Inside the SP class, we declare two basic pointers of "pData" and "reference" type. It is obvious that the "pData" pointer shows the memory location of the actual data of the object that is being reference-counted. In addition, a pointer to a RC class named "reference" points to the class that will store the reference counting information.

Afterwards, we define the constructor of the class SP.

```
SP()
{
    //initialize pData being the null pointer
    pData=0;
    // Create a new reference
    reference = new RC();
    // Increment the reference count
    reference->AddRef();
    //Collect trace
    if(debug) fprintf(stderr,"A!%p!%lu!%d!%p!%s\n",
        reference, sizeof(T),reference->count,
        pData,typeid(T).name());
}
```

The constructor of the class will first initialize the pData pointer to zero and create a new RC object adding one to the reference counting variable. Incrementing the counter at this stage could be translated as a wrong step but it does not make any difference to the functionality of the SP. Last but not least, we can perform the trace collecting process through printing the required information in stderr console output. In terms of programming efficiency, the stderr console might not be the best way to collect the generated traces but it serves our needs.

We collect a number of different information. This trace has:

- The memory address of the Class RC reference.
- The size of the object used as data, thus the size of the type "T" pointer from the template
- The actual reference counting value
- The memory address where the object's data reside, the value of the pData pointer
- The type of the pData object

This constructor can be called by defining an empty SP pointer to an uninitialized object. For example,

SP<MyObject>some_Object;

In addition to creating an empty SP pointer, we include another constuctor for creating a SP pointer to an initialized object.

```
SP(T* pValue) : pData(pValue), reference(0)
{
    //initialize pData to point to the object referenced
    pData=pValue;
    // Create a new reference
    reference = new RC();
```

```
    // Increment the reference count
    reference->AddRef();
    //Collect trace
    if(debug) fprintf(stderr,"B!%p!%d!%lu!%p!%s\n",
                reference,reference->count, sizeof(T),
                pData,typeid(T).name());

}
```

The trace that is collected includes the same information as the trace from the first contructor. The aforementioned constructor can be called as the following example:

SP<MyObject>some_Object=new MyObject();

These are the two ways to instatiate a pointer. Either by instatiating a null pointer with the first contructor or a pointer to an initialized object with the second constructor. We need another constructor to perform a copy operation. In C++, as described in [18], has a special constructor to copy a new object out of an existing one. Most of the times the compiler automatically generates a copy constructor for all classes declared but there are cases in which the programmer needs to define the copy contructor of a class. Thus, the support for user-defined copy constructors is required when an object possesses pointers and non-shareable references. In addition to the copy, the programmer must also define an assignment operator and a destructor.

The copy constructor of an object can be called in case it:

1. is returned by value
2. is passed to a function by value as an argument
3. is thrown
4. is caught
5. is placed in a brace-enclosed initializer list

The source code for the copy constructor is :

```
SP(const SP<T>& sp) : pData(sp.pData), reference(sp.reference)
{
    // Copy constructor
    // Copy the data and reference pointer
    // and increment the reference count
    reference->AddRef();
    //Collect trace
    if(debug) fprintf(stderr,"E!%p!%d!%lu!%p\n",
            reference,reference->count, sizeof(T),pData);
    }
```

This operation will copy the data and reference pointer and increment the count variable. In the end, a trace will be generated containing the following information:
1) The memory address of the Class RC reference
2) The actual reference count value
3) The size of the object used as data, thus the size of the type "T" pointer from the template.
4) The memory address where the object's data reside, the value of the pData pointer

Except for creating and copying, we also have to deal with deleting a SP pointer. The functionality of deleting a SP pointer in the RAII programming idiom is performed by a call to the destructor function. This operation is different from nulling and should not be confused as nulling means assignment and it will be discussed in the following text. Here is the declaration of the destructor of the SP class:

```
~SP()
{
    // Destructor

    //Collect trace incase both intances exits
    if(pData && reference && debug) fprintf(stderr,"F!%p!%p!%u\n",
                                pData,reference, reference->count);
    // Decrement the reference count
    // In case the reference count becomes zero delete the data
    if(reference->Release() == 0)
        {
            delete pData;
            delete reference;
// move out the clause
        }
    }
```

The function collects a trace on the memory address pointed by the pData and the reference(RC class), and the reference count number. Then it decrements the reference count and in case it becomes zero it deletes the data and the reference.

Moreover, we will define functions to overload the basic operators that handle objects. First comes the indirection operator *.

```
T& operator* ()
    {
        if(debug) fprintf(stderr,"G!%p!%p!%u\n",
                    pData,reference,reference->count);
        return *pData;
    }
```

The overloaded function returns a pointer to the pData pointer. Then comes the structure dereference operator ->.

```
T* operator-> ()
    {
        if(debug) fprintf(stderr,"H!%p!%p!%u\n",
                    pData,reference,reference->count);
        return pData;
    }
```

The overloaded function returns the pointer to the data. Both overloaded operators collect traces as the source code shows.

The last step to create a functioning "smart pointer" library is to define an overloaded function for the assignment operator =. But instead of one we define two because of the type difference between SP_null class and the SP class. Thus, we declare the first overloaded function for the SP Class type:

```
SP<T>& operator = (const SP<T>& sp)
    {
        // Assignment operator
        if (this != &sp) // Check for self assignment
        {
            // Decrement the old reference count
            // In case the reference count becomes zero delete
            // the old data and reference information
            // after collecting a trace
            if(reference->Release() == 0)
            {

                if(debug) fprintf(stderr,"I!%p!%p!%u\n",pData,
                                    reference,reference->count);
                 delete pData;
                 delete reference;
            }

            // Copy the data and reference pointer
            // and increment the reference count
            pData = sp.pData;
            reference = sp.reference;
            reference->AddRef();
            if(debug) fprintf(stderr,"J!%p!%d!%lu!%p!%s\n",
                                reference,reference->count,
                                sizeof(T),pData,typeid(T).name());
        }
        return *this;
    }
```

The first step after assigning a new object to a pointer is to decrement the reference counter of the proviously pointed object and in case it becomes zero delete the data and the RC class holding the reference count information. Then, we copy the new data pointer and pointer to the RC class holding the information of the new object incrementing the reference count.

The last declaration for the assignment operator is for assigning a SP_null value to a SP pointer.

```
SP_null& operator = (const SP_null& sp)
 {
      //Assignment operator
      //Collect trace
      if(debug) fprintf(stderr,"K!%p!%d!%d\n",reference,
                        reference->count,reference->count-1 );
      // In case the reference count becomes zero delete
      // the old data and reference information
      // after collecting a trace
      // else delete the pointer to Data
      // and create a new RC class
      if(reference->Release() == 0)
      {
              delete pData;
              delete reference;
      }else {
              pData=0;
              reference=new RC() ;   // or make it zero ??
              if(debug) fprintf(stderr,"L!%p!%d\n",reference,
                                              reference->count );
      }

      return MyNull;
 }
```

3.3 PIN tool

In this section, we use Intel's Pin dynamic binary instrumentation tool to collect memory traces and function calls related to heap allocations and deallocations. This step is required to associate all the memory references and new/delete operations with traces collected from our SP library. The simulation requires both kinds of traces in order to fully leverage the functionalities of a Local Object Store. Thus, our modified tool keeps track of all read and write operations. The logged information are targeted to be associated with the memory references that were caused by our SP library. These memory accesses are not required to take place under the operation of a Local Object Store.

As stated in [24], Pin performs dynamic binary instrumentation on Intel's processors. This enables programmers to develop program analysis tools. During runtime, through an extensive set of API function calls a tool can inject code or retrieve crucial information about the runtime context of the targeted application. All these are done, while keeping away from the programmer all the hidden instruction-set idiosyncrasies. The normal execution flow of the profiled application is maintained because the library saves and restores contaminated registers. Symbol and debug information can be obtained with limited access.

We use and modify an open-source PIN Tool. The atomic memory trace Pin tool written by Steven Pelley can be found online in [25]. The tool except for logging in-order all the memory operations, also includes the function calls to a set of functions defined by the user. We modify this tool to log all the function calls and returns. The essense of these modifications are to define which memory operations will not be required during the operation of a hybrid LOS-Cache memory hierarchy. We present more details on the next chapter when we describe in further detail the implementation of the software stack used for the proof of the concept of a Local Object Store.

Last but not least, we introduce a Region Of Interest function (ROI) that is called as soon as the main function is entered. In C++ there are many static allocations and malloc operations before entering the main function. As we have no control over this code and because a compiler with support for a LOS could produce a totally different source code, we focus on proving the effects of the locality of the references for the regions that we are in control of, thus whatever exists within the main function.

3.4 Microbenchmarks

The next step is to create basic microbenchmarks. We proceed with 3 simple source codes. The following microbenchmarks do not include the SP library. In order to leverage the reference counting information, we just have to change the declarations to instantiate SP objects.

3.4.1 Just Allocate

In this example, we try to evaluate the cost of just allocating a number of objects on the memory configurations that we will investigate.

```
int ParseLine2(char* cstr, int c){
        Person *ab=new Person(cstr,c);
        Person *ac=ab;

        return c;
}
```

The ParseLine2 function is called a number of times to just allocate an object and set an extra pointer directing to it.

3.4.2 SmartDB

The following microbench tries to calculate the average age of a number of Persons that have records in a database. A preconstructed DB file is opened and each line is parsed with a function call.

```
int ParseLine3( string line){

        char *bc= new char[20];
        string *a=new string(line.substr(0,4));
        string *b=new string(line.substr(5,line.length()-4));
        char *cstr = new char[a->length() + 1];
        strcpy(cstr, a->c_str());
        int c=stoi(*b);
        Person *ab=new Person(cstr,c);
        //printf(" c: %s", cstr);
        return ab->age;
}
```

The smartDB example is constructed to examine the case when modularity is used in object-oriented programming languages. Being modular with object-oriented programming means structuring the source code with more objects/classes. Thus, in this microbenchmark the programmer calculates the average (or generally a result) and for that purpose creates some kind of an object.

3.4.3 Stream

Next we formulate a more synthetic microbenchmark that could be used as a better basis to conduct our expirements compared to the previous. The Stream microbenchmark simulates a programming environment in which there exists an input stream of objects. Some of the objects of that are processed will survive and others will just die. It is our expectation that this program has closer to real-life application memory traces.

The source code is depicted in the following text.

```
class  object{
public:
        int param1;
        double param2;
        char str[10];
};

std::vector<object*>  published_stream;

int main(){
        ROI(rand()%10);
        for(int i=0;i<loop/set; i++){
            for(int x=0;x<set;x++){
```

```
                object *a=new object;
                int r=rand()%100;
                if(r>80){
                        a->param1=rand();
                        a->param2=rand();
                        a->str[0]=80;
                        published_stream.push_back(a);
                        a=NULL;
                }
            }

        }
        unsigned long long average=0;
        for(int x=0;x<published_stream.size();x++){
                        average+= published_stream[x]->param1;
            }
 }
```

The aforementioned stream is called "published_stream". Sets of ten objects, each assigned with a randomly generated number from 1 to 100. We let objects with a number higher than 80 to survive in the "published_stream". In case we set the "loop" with a high number, we expectation that only 20% of the objects to survive. With a small "loop" number, the result is obviously unpredictable. Due to space limitations for the produced traces we define the "loop" number to be 20.

Microarchitecture Investigation 4

4.1 The architecture of a Local Object Store

In this Chapter, we present the architecture of a Local Object Store. The two main modules are the UID lookup table and the data entries for the fields of the objects along with the tags. The UID Lookup table should be closer to the Memory Management Unit(MMU) of the processor in order to serve faster the object related operations.

The UID Lookup table contains the translations of UIDs to local indices. This module is required in order for the MMU to know if an object is located either in the main memory or in the LOS-Data Array.

The LOS-Data Array contains fixed-sized entries, thus the maximum number of fields for the objects that can be allocated within a LOS is fixed. We investigate the potential problems of this design decision in the following sections.

Of course, the LOS unit requires a Control Unit (CU) and an external bus to communicate with the MMU. The Control Unit receives and executes all the LOS instructions.

A Local Object Store is responsible for mainly:

- Allocation of an object
- Deallocation of an object
- Field Access, Read or Write
- Purging of a local object to main memory

These operations are orchestrated by the CU of the LOS in coordination with the UID Lookup Table and the tags of the . The following figures depict the basic required components.

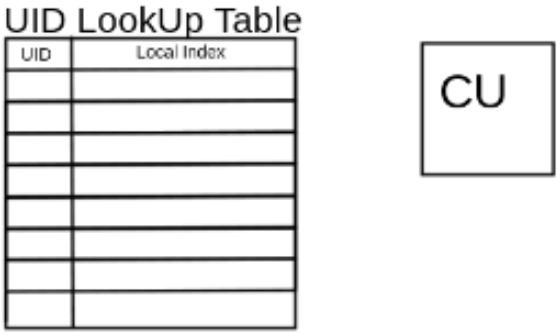

Figure 4.1: The UID LookUp Table and the Control Unit of a Local Object Store

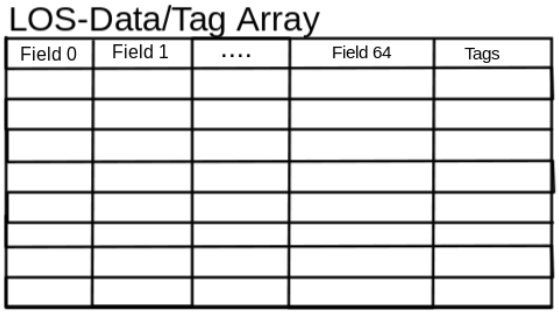

Figure 4.2: The Data and Tag Arrays of a Local Object Store

4.2 The Functionalities of a Local Object Store

In this section, we present how the Control Unit performs all operations in detail. These operations require the extension of the Instruction Set Architecture in order to be served. Precisely, four extra instructions are required, one for each operation.

In the following subsections, we describe each operation from the executed instruction to the logical operational flow between:

- The processor
- The Memory Management Unit
- Cache Hierachy
- The Local Object Store Unit and its internal structures

4.2.1 Read Field - Access

We extend the ISA of the processor with the read field instruction:

RFLD #UID, #FIELD, #SIZE

The request includes the UID that is the memory address where the object starts or it is translated to a local index, the field number and how many bytes the program requests. Firstly, a "RFLD" operation is to be executed in the pipeline of the processor. All arguments are passed to the MMU. The MMU uses the UID to search if it is located inside the LOS structure. In parallel with the entry check of the table, an addition of the UID and the field number is performed. In case the object has been purged to the main memory the address will be ready as soon as the lookup request in the UID Lookup table has finished. In case the object is located in the LOS structure, the index, the field and size parameters are passed to the Control Unit of the LOS.
Now, the LOS is responsible for accessing the field and reading the appropriate size of data. The Control Unit receives the operands of the instruction and is now responsible to manage all internal signals. Last but not least, the data have to be committed back on the external bus to either the MMU or directly to the processor that requested the service.

4.2.2 Write Field - Access

For the write operation we follow the same procedure. Again, we include a new instruction in the ISA of the processor, now with a write field instruction:

WFLD #UID, #FIELD, #DATA

The request includes the UID that is either the memory address where the object starts or it is translated to a local index, the field number and datum to be written. A "WFLD" operation is to be executed in the pipeline of the processor. All arguments are passed to the MMU. The MMU uses the UID to search if it is located inside the LOS structure. Again, in parallel with the entry check of the table, an addition of the UID and the field number is performed. In case the object has been purged to the main memory, the address will be ready as soon as the lookup request in UID Lookup table has finished. And a common memory access will be executed. In case the object is located in the LOS structure, the index, the field and the datum are passed to the Control Unit of LOS.
Now, the LOS is responsible for accessing the field and writing the appropriate data to the right data entries.

4.2.3 Object Allocation

We modify the *new* operation for the purpose of allocating an object in the LOS unit. To better understand, we present the steps that are followed during the execution of a *new* operation in a C++ program after we state the different ways to allocate an object. Memory in C++ is divided into two main categories:

- The Stack: where all variables declared in a function will take up memory space
- The Heap: where the program can dynamically allocate memory space

Generally, it is possible to allocate objects in the stack, but in order to dynamically allocate memory space in the heap, the *new* operation or the malloc function must be called. Allocation in the heap is a two step process:

- Firstly, raw memory is requested from the Operating System
- Once that memory is granted, the new object is constructed in it by a call to the object or class constructor

Precisely, the initial step does not involve a system call each time the *new* operation is called. The *new* operator denotes a request for memory allocation on a process's free store. In addition, a different low-level mechanism is used to allocate pages of memory. On the Windows platform the mechanism is the VirtualAlloc(). In POSIX-compliant Unix based operating systems the Mmap() is used.

As stated in [26], Mmap can be used to map a user predefined memory space to a file. It has a "lazy" read operation. Initially, the data in the targeted file are not accessed from the secondary storage devices and also do not occupy physical space in the main memory. Thus, accessing the hard-disks follows after a request to a particular

address is issued. Therefore, Mmap performs *demand paging* in a natural manner. The program must release the ownership of the requested memory-mapped address space after completing all operations related to that working set.

The memory allocation algorithm, thus the implementation of *new* operation and *malloc*, serve allocation requests by either assigning a position within the already acquired memory pages or requesting a new page from the Operating System through a *Mmap()* call.

We present the description of the mmap function from Linux Programmer's Manual [27]:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
int munmap(void *addr, size_t length);
```

" mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping.

If *addr* is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping), are initialized using length bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by sysconf.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). "

We provide an extension to the virtual address space by introducing the LOS unit. Generating an UID to be used for a new entry in the UID Lookup table requires the modification of the algorithm responsible for the management of the free store. More work needs to be done in order for a compiler to control the memory pages that are associated with the dynamic allocations that take place within the LOS unit. It is our expectation that it is possible for the allocation of the space and the constructor to be executed in parallel due to the fact that the Local Object Store can be thought of as a huge register file that is linked only through the UID LookUp table to the different places of the virtual address space that represent the objects. Thus, the processor could use the local index as a reference to the object till the UID is resolved. Moreover, to fully simulate an environment with memory management of the virtual address space, we need to make a full system simulator. Possibly, an alternative would be to use one of the open source full system simulators and modify it. Although, the results would be more accurate, considering the time limitations we do not follow this approach. We expect to develop a basic proof of concept for the benefits of using a Local Object Store in a more constrained simulated environment.

First, the modified *new* operation generates an UID for the object to be allocated checking if it is under the fixed size of a LOS entry. After that the memory allocation request is passed to the LOS unit. We extend the ISA of the processor with the new

LOS Object Allocation instruction:

NOBJ #UID, #SIZE

The request includes the UID that is the memory address where the object starts and the size information. The processor passes these arguments to the control unit of the LOS. The LOS unit now is responsible to find a free entry either by using an available one or by purging an object to the main memory. For the simulation we use a Random Replacement Policy but further work could reveal more useful and cost effective alternatives. Last step is a call to the constructor of the class.

4.2.4 Object Deallocation

Furthermore, we require a special instruction so that the program can know if an object has to be hardware deallocated or a normal delete operation has to be called. As described in [43], generally, after an application finishes with the usage of an object, it has to release the ownership of the memory space acquired. This is an essential responsibility of the application towards the operating system. Thus, the operating system can give the ownership of that address space to another program/process. Sometimes, object destruction does not happen, for example in case the underlying mechanism does not require extra memory space or in case the application has a short execution time. Most of the times, especially for programming languages that use a garbage collector to manage their heap space, object destruction just means memory deallocation. Depending on the memory management system used, the are cases that extra operations must be performed before the deallocation. For instance, in languages with manual memory management the destructor of a class/object must destroy all member classes/objects. Another example is on reference-counted objects that the hidden implementation has to decrement the counters to check if deallocation should occur. We try to eliminate that extra software overhead by serving the deallocation to our Local Object Store.

We extend the ISA of the processor with the new LOS Object instruction

DOBJ #UID

So that in case the delete operation finds the object located in the LOS structure the following steps will be performed:

- The local index of the entry will be marked available
- The UID Lookup Table has to find and delete the entry

4.2.5 Purging a local object

This operation can happen when an object allocation request is issued for the LOS unit and there is no available entry. We define the following steps to be performed when a the Control Unit of the LOS purges an object.

- The object is written back to main memory

- The local index of the entry will be marked available
- The UID Lookup Table has to find and delete the entry

4.2.6 Reference Counting

In order to be able to manipulate on reference-counted objects, a Local Object Store should know how many pointers are referencing the local object. Thus, special instructions have to be issued from the processor in case the application creates a new pointer and assigns to it the value of an old one.

We include two instructions for the purposes of increasing and decreasing the counter of reference.

ADDREF #UID

DECREF #UID

These operations are issued by the application. Thus, it is essential to provide compiler support to utilize a Local Object Store. After the commands are executed in the pipeline of the processor, the locality of the object is checked in the UID Lookup Table.

4.3 Software Architecture

Initially, we explore modularly the required software components and integrate them into a final version of the MMU-LOS integrated Simulator. The main components of this thesis project are:

- Trace Pin tool: Modified pin tracing tool found in [25]
- Basic MMU Simulator
- MMU - LOS Simulator

We develop all the aforementioned software components in C++. First step that we follow is to modify the atomic-trace-tool, a PIN-tool to produce multi-threaded atomic memory traces. The source is available for public use in [25]. This tool provides atomic instrumentation by simulating cache coherence. In addition, the tool will trace thread start/end, an optional region of interest, and user-provided function calls and returns. The primary alternative to this tool is architectural simulation. Most simulators are complicated to learn as well as to use because getting OSes and workloads running properly may be difficult, and slow due to the fact most simulators are single threaded and cannot leverage multi-threading to produce a faster trace/simulation. The memory traces are written in a output file. Each line contains one event as a tab delimited list. All entries start with a time-stamp and threadid:

- Memory access: Each instruction may read two addresses and write one. There are possible sub-entries for each of these accesses. The second read does not contain a size field as it may only occur with the first read and have the same size (that is,

r2's size is the same as r's).
<timestamp><thread>m [r <address><size>] [r2 <address>] [w <address><size>]

- Thread registered:
<timestamp>threadid tr
- Thread finished:
<timestamp>threadid tf
- Function call: All requested functions are traced, even if not on a registered thread. The first 3 arguments of the function are traced as well as the stack pointer to match up calls and returns
<timestamp><threadid>fc <function name><function stack pointer><first arg><second arg>
- Function return:
<timestamp><threadid>fr <function name><function stack pointer><return value>
- start Region of Interest:
<timestamp><threadid>start_roi
- end Region of Interest:
<timestamp><threadid>end_roi
- context change: Context changes may interrupt the locking necessary to provide atomic tracing. On a context change consider that the next access may not be traced atomically. – It is unclear how the PIN internals work and if this is really a concern (I haven't observed any context changes yet).
<timestamp><threadid>ctxt_change
- thread sync: When threads flush other threads to keep all threads close in timestamps the merging process must be made aware of this. <timestamp><threadid>thread_sync

In addition to memory accesses many functions are traced. A few are specific to this tool, but any user-provided function can be traced. We modify this functionality and trace all functions. As a secondary step during parsing we check exactly what is required for the simulation. Moreover, through this modification we are able to determine which functions have a return call and which do not. This is a essential step in order to justify which memory operations can be saved during the operation of the Local Object Store.

Due to unit testing issues we proceed with a simpler choice as a start to integrate the tracing and the simulation proccess. We mainly use the aforementioned Pin tool to collect traces. In addition, we develop two C++ based simulators. One simulator for the common Memory hierarchy and one modified one to include the LOS unit.

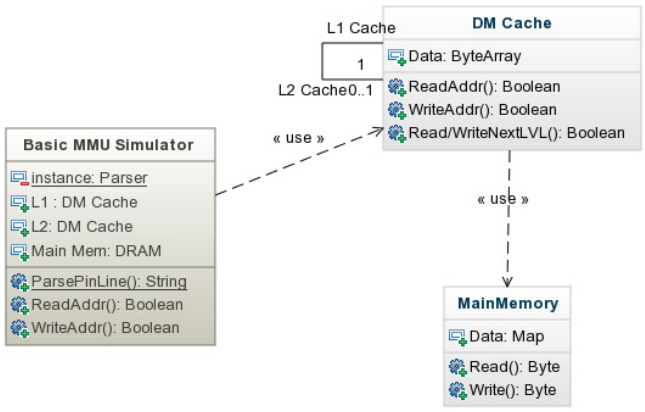

Figure 4.3: A Software Architecture for the the basic MMU simulation

As the pin tool collects the data into a file, our initial step is to develop a parser. Before the parser starts reading the trace file, the program instantiates all required components:

- Level 1 Directly Mapped Cache - Write back policy
- Level 2 Directly Mapped Cache - Write back policy
- Local Object Store
- Main Memory – We use a Std::Map (addr,byte) for the mapping of address to data

The simulator represents a simple Memory hierarchy but without the functionality of the Translation Look-aside Buffer. It is out of the scope of this Master thesis to define a full system simulator. Moreover, we do not expect any drastic changes in the outcome of this simulation platform in case Translation Lookaside Buffers (TLB) are introduced. It is our expectation that the TLB simulation would provide a more accurate result in terms of timings of read and write operations to the Main Memory, but not in terms of the locality of the references.

4.4 Results for an initial study case

In the beginning of the following section, we present the results of the offline Simulation. We use the three microBenchmarks of chapter 3 to generate traces with our modified Pin tool and store them into files. The trace files are used as an input for the offline Simulator. We use 4 scenarios for the simulation, under which we evenly split the size of the Level 1 cache of the cache-only hierarchy into a Level 1 cache and a LOS for the hybrid organization and double their memory space in each scenario. We use the same size for the Level 2 cache for both configurations, also doubling its space in each scenario. Due to software limitations we do not simulate beyond the size of a 128KB Level 2 cache memory.

4.4.1 Just Alloc

The microBenchmark is a simple example of allocating a high volume of objects. We present the simulation results for different configurations using our first example with the Region Of Interest (ROI) function, to limit our scope in the main function of the application.

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	1184	2341192	233	971104
4-4-64	907	2341469	169	971169
8-8-128	701	2341675	107	971230
16-16-128	497	2341879	71	971266

Table 4.1: Level1 Cache Traffic with LOS – Just Alloc With ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	568	849	17466	962054
4-4-64	520	556	17425	953647
8-8-128	478	330	17339	942981
16-16-128	444	124	17227	927221

Table 4.2: Level2 Cache Traffic with LOS – Just Alloc With ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocations	L2 Write hit	Field Writes	Purge Count
2-2-32	18034	1113728	10000	962054	880000	9968
4-4-64	17945	1087296	10000	953647	880000	9932
8-8-128	17817	1023104	10000	942981	880000	9872
16-16-128	17671	1012160	10000	927221	880000	9744

Table 4.3: LOS and Main memory and traffic – Just Alloc With ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	54721	4715157	22840	3993889
8-64	9249	4760629	20225	3996504
16-128	5425	4764453	18917	3997812
32-128	3560	4766318	18277	3998452

Table 4.4: Level1 Cache Traffic – Just Alloc With ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	21837	55724	0	2388224	21837	1204864
8-64	20534	8940	0	1464128	20534	1139712
16-128	19793	4549	0	1295552	19793	1061184
32-128	19793	2044	0	1204864	19793	1061184

Table 4.5: Level2 Cache and Main Memory Traffic – Just Alloc With ROI

On the following chart we quantitatively present the benefits of incrementally doubling the sizes of the Level 1 and 2 caches as well as the LOS in both configurations. We observe that including a LOS structure, we reduce the accesses for both operations more than quadrupling the size of the caches. Thus, we validate the claims of [37] on the benefits of hybrid memory hierarchies.

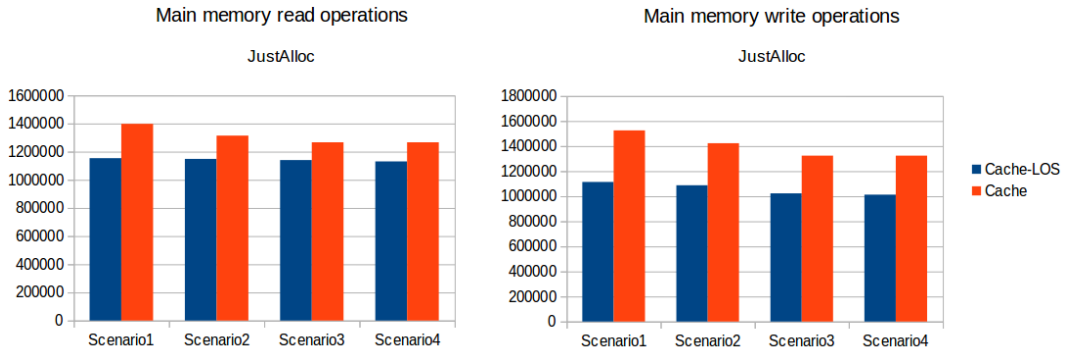

Figure 4.4: Reduction in main memory read/write operations for Just Alloc with ROI under cache-only and cache-LOS configurations

On the next graph, we depict the advantages of using a hybrid memory architecture with an integrated LOS unit with respect to a cache-only configuration. The main memory traffic is the dominant cost in modern computer systems. Saving data movements from the CPU to the main memory decreases power consumption and increases performance. Moreover, allocation is the basic operation related to objects. We expect that by introducing this microBenchmark to evaluate the memory-based performance will give us a feedback on real-life applications that go through phases of allocating fast a big number of small objects. This is a common behavior for graph traversal algorithms and many other modern programs.

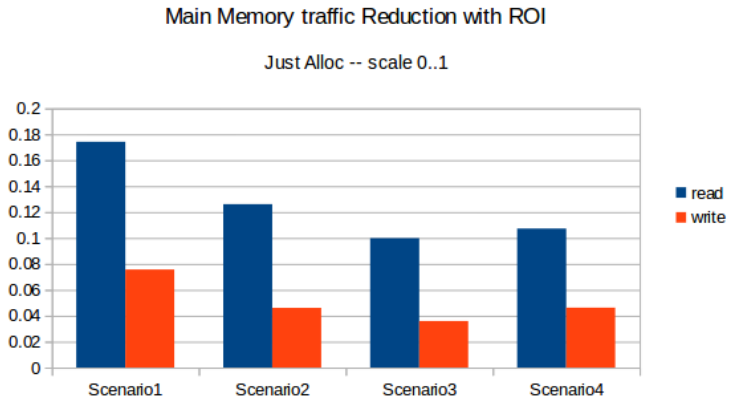

Figure 4.5: Main Memory Traffic Reduction for Just Alloc with ROI

The performance drops almost by 50% without the use of ROI. We consider this to be of less importance due to the fact that before entering the region of interest, thus the main function, the compiler is responsible for the static allocations required. We include the tables with the simulation results in the appendix.

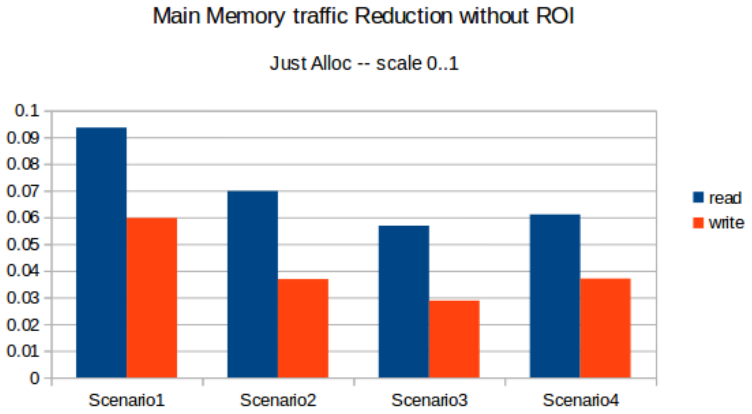

Figure 4.6: Main Memory Traffic Reduction for Just alloc without ROI

Both graphs appear to depict a contradiction to the fact that increasing the size of both memory structures should increase the main memory reduction. Thus, moving from scenario 1 to scenario 4 we should observe an increase in the percentage, meaning increased benefits of using a LOS. Obviously, we observe a degradation in the gained benefits. Moreover, the example can obviously operate during its whole execution locally in the LOS, therefore realistic results would be really much higher than the ones that we obtain. There are two main reasons for the aforementioned behavior. First of all, due to the small working set of the application in the first scenario (2/2/32 KB) the hybrid structure has already provided a better hit rate for Level 2 cache which is almost equal to the amount of reduction in main memory references when we double the sizes of the caches from scenario 1 (4/32 KB) to scenario 2 (8/64 KB). Thus, in the first scenario due

to the simplicity of the application the benefits of using a hybrid configuration are greater than even doubling size of the structures. This can be related to gaining better locality by using more associativity in a cache than doubling the size due to the application's memory access patterns. Moreover, since we have not yet included in the parser of the input files the ability to associate Pin and SP traces, we generate main memory traffic that is obviously not required. This is based on the fact that the program instantiates an object which immediately dies when the scope is exited, therefore all purges could have been avoided.

In simple words, due to the small working set of the application and lack for support on reference counting by the parser, the Level 2 Read misses that are saved by the integration of a LOS are converted into Write Misses from the purges of the objects.

4.4.2 SmartDB

The microBenchmark is the simple example on basic manipulations of objects. We present the simulation results for different configurations with the Region Of Interest function.

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	891632	36303997	127941	19129543
4-4-64	647323	36502135	64249	19188532
8-8-128	276157	36848880	2647	19247331
16-16-128	226901	36882215	1822	19245592

Table 4.6: Level1 Cache Traffic with LOS – SmartDB With ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	128191	891382	133006	52411622
4-4-64	20546	691026	100901	30732225
8-8-128	14041	264763	87583	16060506
16-16-128	16186	212537	84355	11325069

Table 4.7: Level2 Cache Traffic with LOS – SmartDB With ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocs	LOS Deallocs	Field Reads	Field Writes	Purge Count
2-2-32	261197	8915840	140003	3	1137003	2685865	139970
4-4-64	121447	6638208	140003	3	1183174	2690568	139938
8-8-128	101624	5884096	140003	3	1207595	2693371	139874
16-16-128	100541	5844800	140003	3	1223516	2695935	139746

Table 4.8: LOS and Main memory and traffic – SmartDB With ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	1090078	70294803	419491	51770190
8-64	549434	70835447	198076	51991605
16-128	440078	70944803	184667	52005014
32-128	204465	71180416	138045	52051636

Table 4.9: Level1 Cache Traffic – SmartDB With ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	342510	1167059	0	37514496	342510	10786752
8-64	197457	550053	0	20171648	197457	8856192
16-128	186826	437919	0	18148032	186826	8539072
32-128	186826	155684	0	10786752	186826	8539072

Table 4.10: Level2 Cache and Main Memory Traffic – SmartDB With ROI

On the next chart, we quantitatively present the benefits of doubling of the sizes of the Level 1 and 2 caches as well as the LOS in both configurations and under all scenarios. The SmartDB microBenchmark does not exhibit the same amplitude of improvement in memory access-based performance compared to the previous example. Nevertheless, we observe a decrease in the number of read operations by the integration of a LOS under the specified scenarios. Moreover, we notice an increase in the write operations due to the huge number of purges the LOS performs that pollute the Level 2 caches. We expect that with the completion of our infrastructure this example would reveal better results, for the reason that the instantiated objects die after the function returns.

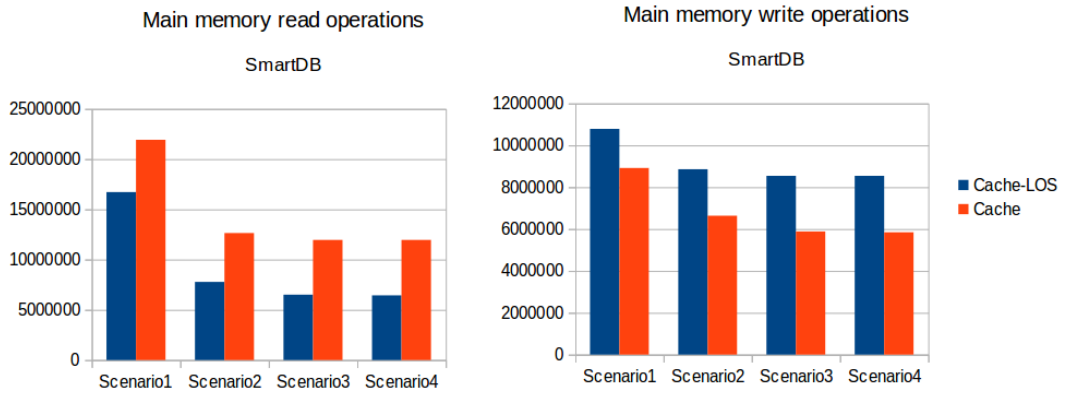

Figure 4.7: Reduction in main memory read/write operations for SmartDB with ROI under Cache-only and Cache-LOS configurations

On the next graph we depict the percentages in traffic reduction from and to the main memory from the use of our proposed hybrid configuration, compared to a cache-only memory subsystem.

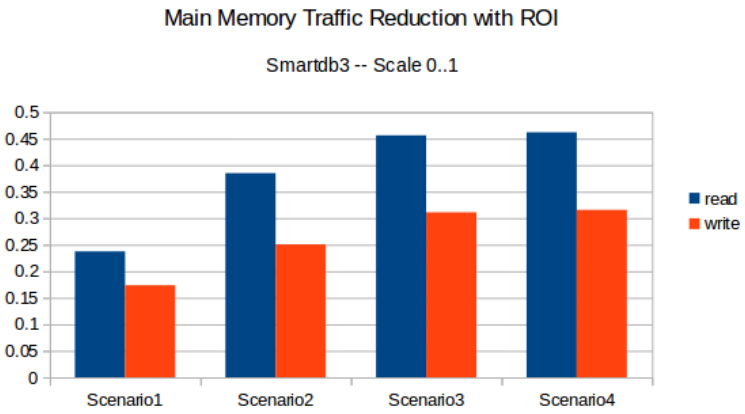

Figure 4.8: Main Memory Traffic Reduction for SmartDB with ROI

For SmartDB we show that there is not a big difference between the use of ROI and not using it. This is mainly due to the fact that there is a big number of iterations over the basic loop. Thus, the ROI has a huge number of references compared to the number of memory references in the ROI of the previous example.

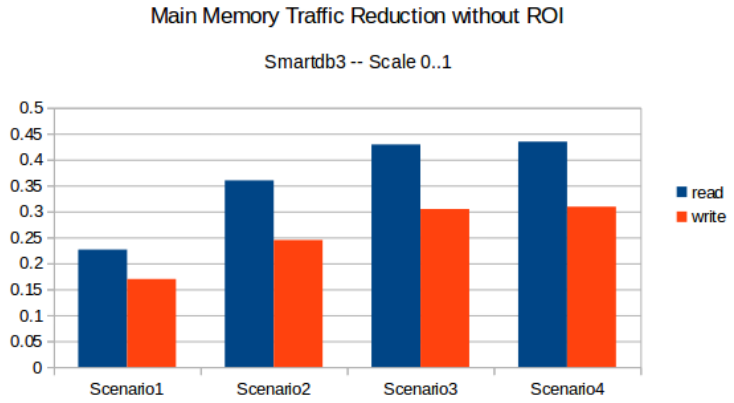

Figure 4.9: Main Memory Traffic Reduction for SmartDB without ROI

The graphs show that the integration of the LOS structure reduces main memory traffic compared to a cache-only memory architecture. This observation is logical since using a bigger memory space for the objects results in less evictions. Last but not least, we observe an increased pressure on Level 1 and Level 2 caches that originates from the purges the LOS performs.

4.4.3 Stream

The microBenchmark is the simple example on basic manipulations of object streams. We present the simulation results for different configurations and with the Region Of Interest function.

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	1487	41223	215	25098
4-4-64	1150	41560	150	25163
8-8-128	824	41886	99	25214
16-16-128	546	42164	57	25256

Table 4.11: Level1 Cache Traffic – Stream With ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	471	1231	0	23872
4-4-64	372	928	0	18688
8-8-128	267	656	0	12800
16-16-128	267	336	0	8064

Table 4.12: Level2 Cache Traffic – Stream With ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocs	LOS Deallocs	Field Reads	Field Writes	Purge Count
2-2-32	471	7168	29	4	116	169	0
4-4-64	372	6336	29	4	116	169	0
8-8-128	267	2752	29	4	116	169	0
16-16-128	267	2752	29	4	116	169	0

Table 4.13: LOS and Main memory and traffic – Stream With ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	2984	111192	258	34880
8-64	2341	111835	199	34939
16-128	1895	112281	131	35007
32-128	1736	112440	120	35018

Table 4.14: Level1 Cache Traffic – Stream With ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	1856	1386	0	29120	1856	17792
8-64	1747	793	0	22656	1747	22464
16-128	1453	573	0	15680	1453	16896
32-128	1453	403	0	17792	1453	16896

Table 4.15: Level2 Cache and Main Memory Traffic – Stream With ROI

On the following chart, we quantitatively present the benefits of doubling of the sizes of the Level 1 and 2 caches as well the LOS in both configurations and under all scenarios. We observe the same scaling but higher amplitude in the reduction of the main memory accesses compared to the SmartDB microBenchmark. This is based on the fact that the huge trace files that Stream produces do not utilize the full memory space of the LOS. Thus, we require an online simulator to use a large number of objects for the LOS to start evictions. We do not expect better results for this microBenchmark with

the incorporation of the reference counting functionality to the simulator. Moreover, an online simulation would have more realistic results.

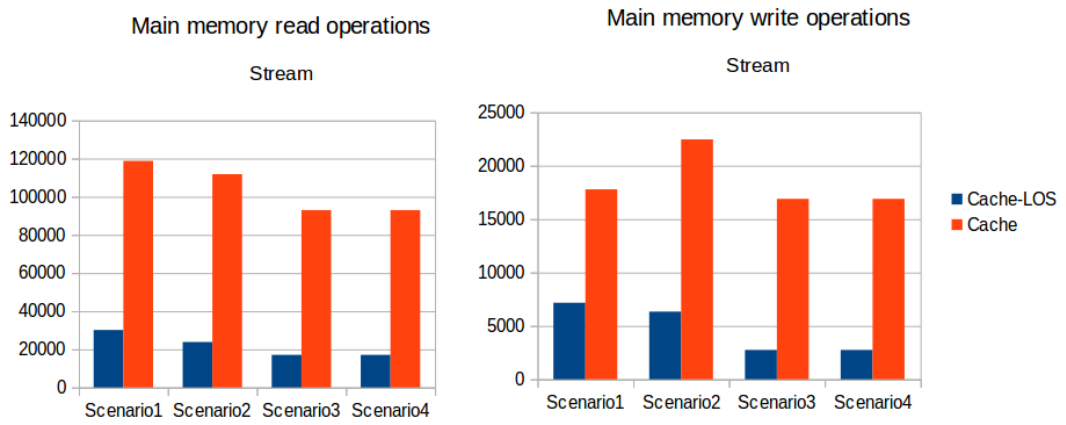

Figure 4.10: Reduction in main memory read/write operations for SmartDB with ROI under Cache-only and Cache-LOS configurations

The next graph shows again a beneficial usage of the LOS unit.

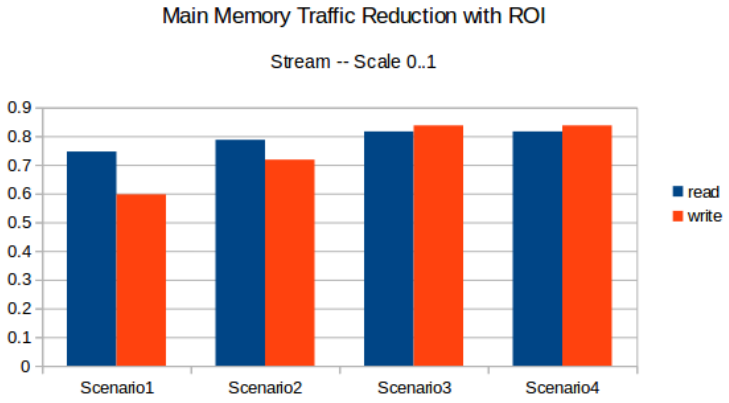

Figure 4.11: Main Memory Traffic Reduction of Stream with ROI

Moreover, as we depict in the next graph, there is a big difference between using and not using a ROI. This is due to the small size of the object used. We need a lot of iterations to produce enough traces to overcome the number of the static allocations before the main function.

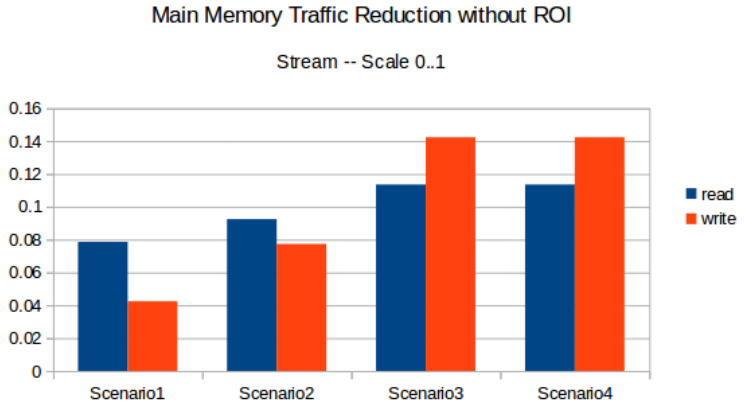

Figure 4.12: Main Memory Traffic Reduction of Stream without ROI

The results reveal the same situation that we observe in the previous example. Therefore, the integration of the LOS decreases the number of references to the main memory compared to a cache-only hierarchy. Moreover, this example shows the highest improvement. This is possible due to the fact that all instantiated object live inside the Local Object Store and none is purged back to main memory.

5 Conclusions & Future Work

In this chapter, we present our conclusions based on the results that we obtain through the simulation of a memory architecture that includes a Local Object Store. Moreover, we present the limitations of our proof of concept and describe the range of benefits. Compared to a cache, the main target goal for a LOS is to exploit the locality of short-lived objects by using an extension of the address space for their allocation and destruction, while compacting on-the-fly the used address space. In the end, we recommend possible future directions that will guide us into fully defining the advantages and disadvantages that an integration of a Local Object Store with modern memory architectures would bring to a computer system.

5.1 Summary

We formulated the hypothesis that dynamic heap allocation has the potential to enhance the main processor's functionalities, and made the assumption that the concept of *object caching* is advantageous for a modern memory architecture due to the dropped IC costs. To prove our hypothesis, we started with the modification of the HPROF profiling tool, we quantified the potential benefits of a memory architecture with an integrated LOS by counting the percentage of objects that could be allocated inside our proposed design in a set of Java Benchmarks. We continued with the implementation of our "smart pointer" library that can trace the required information on reference-counted objects for the C++ programming language. Our library enabled us to get detailed information in order to simulate the functionalities of a LOS compared to the modified HPROF tool. In addition, we developed a set of three microBenchmarks written in C++. Moreover, we used and enhanced a Pin tool to keep track of all function calls and returns, as well as memory accesses in the aforementioned microBenchmarks. Afterwards, we proposed a microarchitecture for a LOS and required extensions to the instruction set architecture of the main processor. Using the traces of our microBenchmarks, we simulated and compared a Cache-LOS memory architecture with a cache-only organization, proving the benefits in terms memory access-based performance of the integration of a LOS with a modern memory architecture.

5.2 Conclusions

Motivated by an extensive literature search we have proposed an architecture for a Local Object Store and developed a simulation infrastructure for evaluating its performance as compared to a conventional cache. We started our research by identifying the differences between using a scratchpad memory and a cache memory and reached an interesting

conclusion. **It is more beneficial, in terms of the locality of memory references, to use cache memories for spatially referenced data structures, while scratchpad memories serve better references that exhibit temporal locality.** The operational costs for checking tags and the flat address space of a cache seem ideal for structures as arrays, which most of the times exhibit spatial locality in their access patterns. On the contrary, the overhead for software managing the private address space of a scratchpad memory, enables programmers to arrange the data transfers according to their applications runtime characteristics. Thus, scratchpad memories reveal great opportunity for temporal locality references, because the application can evict data that are not required for the future execution flow. In relation to that, when objects are mapped to caches, since the main memory requests are done in cache lines, caches pull data adjacent to objects into the cache, whereas a LOS has the benefit that short-lived objects do not create main-memory pressure by leveraging their temporal locality of references to fields and the ability to organize LOS's private memory space. Our conclusion is that most modern computer system would benefit from incorporating a cache and a scratchpad memory. In addition, we focus on objects based on the fact that accesses to object fields exhibit temporal behavior.

Moreover, another reason that we focus on objects is their "infant mortality" property. In Chapter 1, we mention the fact that most modern applications, written in object-oriented programming languages, produce many short-lived objects. Thus, our literature research guides us into using garbage collection information to enhance the knowledge of the processor on the locality of objects. In Chapter 2, we conclude by showing that a high percentage of objects in today's real-life applications can be allocated within a Local Object Store. Thus, we partially validate our findings in Chapter 1. The results in Chapter 2 validate our intuition that we should use Java as our first target language for the framework for simulating the LOS functionalities. Although, we provide a basic framework to use Java, we switch to C++ due to time limitations in developing the software for tracking the real lifetime of heap-allocated objects.

We have evaluated our design on an initial set of microbenchmarks and conclude that on a configuration where the Level 1 cache is split evenly between a LOS and a conventional cache and shown a 39% on-average benefit in main-memory bandwidth, which we believe is a key limiter in modern multicore processors. The programs, that we formulate, are basic corner cases on operations for object manipulation that exist in modern applications.

5.3 Future Work

We recommend Java to be the main target programming language for the pursued research directions. This is based on our findings in Chapter 2, where we show that a high percentage of the objects allocated by the programs of the DaCapo Benchmark suite would fit within the entries of a Local Object Store. In addition, the benefits for Java can be expected to be larger than those for C++ because the Java runtime system, the details of which are hidden from the user, takes care of object management and Java programmers therefore tend to make more extensive use of object-oriented features. Therefore, we recommend as future work the development of our modified Hprof tracing

tool based on JVM TI that will capture all the information that is required for the proper functionality of the reference counting system of a Local Object Store. This implies of keeping track of all pointers instantiated in the stack as well as following the full garbage collection root set. Although the functionalities exist, due to time limitations, our parser of the input trace files does not yet associate the traces from our SP library with the traces of the modified Pin tool. On the other hand, we anticipate a small number of extra main memory references compared to the results of our simulator, when the *new* operation is redefined for compiler support to a Local Object Store.

In order to produce a more realistic simulation environment, we must develop a full system simulator and a compiler that supports the functionalities of a Local Object Store. Moreover, our simulator must be extended with the functionalities of associating the SP and the Pin traces. In addition, we suggest the integration of the tracing tools with the simulator to produce an online simulator for a Cache-LOS memory organization. Last but not least, a Local Object Store has significant potential in exploring the software and hardware cooperation in performing memory management operations. Especially, for the *new* operator there are many alternative paths that can be followed. We believe that with this master thesis we develop a basic framework up on which we can exploit all the trade-offs for improving the division between hardware and software responsibilities in memory management techniques.

Bibliography

[1] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 1 (March 1995), 20-24. DOI=http://dx.doi.org/10.1145/216585.216588

[2] Chapter 12.5 Creation of New Class Instances of the Java language specification, available from: http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.5 (Accessed 7-Dec-2015)

[3] Garbage collection roots available from: http://www-01.ibm.com/support/knowledgecenter/SS3KLZ/com.ibm.java.diagnostics.memory.analyzer.doc/gcroots.html (Accessed 7-Dec-2015)

[4] Lecture presentation for CMSC 433 Programming Language Technologies and Paradigms Spring 2006. University of Maryland, College Park. (Accessed 7-Dec-2015) http://www.cs.umd.edu/class/spring2006/cmsc433/lectures/gc.pdf

[5] Article on Garbage collection available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29

[6] Article on Reference counting available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Reference_counting

[7] Steve Yegge's speech at Stanford's EE Computer Systems Colloquium on "Dynamic Languages Strike Back" available from: (Accessed 7-Dec-2015) http://steve-yegge.blogspot.nl/2008/05/dynamic-languages-strike-back.html

[8] Article on Tracing garbage collection, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Tracing_garbage_collection

[9] The Java Virtual Machine Tool Interface documentation, available from: (Accessed 7-Dec-2015) http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[10] TIOBE Index for December 2015 available from: (Accessed 7-Dec-2015) http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[11] Tal Weisse's article on "Garbage Collectors Serial vs. Parallel vs. CMS vs. G1 (and whats new in Java 8)" http://blog.takipi.com/garbage-collectors-serial-vs-parallel-vs-cms-vs-the-g1 -and-whats-new-in-java-8/

[12]Stroustrup, B. (6 May 2014). "Lecture:The essence of C++. University of Edinburgh.", available from: (Accessed 12-Jun-2015) https://www.youtube.com/watch?v=86xWVb4XIyE

[13] Article on object storage, available from:(Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/C%2B%2B#Object_storage

[14] Article on the concept of "Resource Acquisition Is Initialization" available from:(Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

[15] Stroustrup's answer on "Why doesn't C++ provide a "finally" construct?": available from:(Accessed 7-Dec-2015) http://www.stroustrup.com/bs_faq2.html#finally

[16] Article on smart pointers, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Smart_pointer

[17] Microsoft's Developer Network documentation on "Smart Pointers (Modern C++)", available from: (Accessed 7-Dec-2015) https://msdn.microsoft.com/en-us/library/hh279674.aspx
[18] Article on the Copy constructor, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Copy_constructor_%28C%2B%2B%29
[19] Whitham, J.; Schoeberl, M., "WCET-Based Comparison of an Instruction Scratchpad and a Method Cache," in Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on , vol., no., pp.301-308, 10-12 June 2014 doi: 10.1109/ISORC.2014.48
[20] Seager, K.O.; Tiwari, A.; Laurenzano, M.A.; Peraza, J.; Cicotti, P.; Carrington, L., "Efficient HPC Data Motion via Scratchpad Memory, " in High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: , vol., no., pp.801-805, 10-16 Nov. 2012 doi: 10.1109/SC.Companion.2012.111
[21] Puaut, I.; Pais, C., "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07 , vol., no., pp.1-6, 16-20 April 2007 doi: 10.1109/DATE.2007.364510
[22] Yu Liu; Wei Zhang, "Exploiting multi-level scratchpad memories for time-predictable multicore computing," in Computer Design (ICCD), 2012 IEEE 30th International Conference on , vol., no., pp.61-66, Sept. 30 2012-Oct. 3 2012 doi: 10.1109/ICCD.2012.6378618
[23] Chakraborty, P.; Panda, P.R., "SPM-Sieve: A framework for assisting data partitioning in scratch pad memory based systems," in Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on , vol., no., pp.1-10, Sept. 29 2013-Oct. 4 2013 doi: 10.1109/CASES.2013.6662527
[24] Intel's Developer Zone, website for the Pin tool, available from: (Accessed 7-Dec-2015) https://software.intel.com/en-us/articles/pin-a-dynamic-binary -instrumentation-tool
[25] Repository of Steven Pelley's atomic memory trace tool: https://github.com/stevenpelley/atomic-memory-trace (Accessed 7-Dec-2015)
[26] Article on the Mmap function, available from: https://en.wikipedia.org/wiki/Mmap (Accessed 7-Dec-2015)
[27] Chakraborty, P.; Panda, P.R., "SPM-Sieve: A framework for assisting data partitioning in scratch pad memory based systems," in Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on , vol., no., pp.1-10, Sept. 29 2013-Oct. 4 2013 doi: 10.1109/CASES.2013.6662527
[28] Duncan Coutts; Andres Loh "Deterministic Parallel Programming with Haskell", Computing in Science and Engineering, vol. 14, no. 6, pp. 3643, Nov.Dec. 2012, doi:10.1109/MCSE.2012.68
[29] Chang, J.M.; Gehringer, E.F., "Object caching for performance in object-oriented systems," in Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on , vol., no., pp.379-385, 14-16 Oct 1991 doi: 10.1109/ICCD.1991.139924
[30] David R. Ditzel and David A. Patterson. 1998. Retrospective on high-level language computer architecture. In 25 years of the international symposia on Computer architec-

ture (selected papers) (ISCA '98), Gurindar S. Sohi (Ed.). ACM, New York, NY, USA, 166-173. DOI=http://dx.doi.org/10.1145/285930.285976
[31] Article on the Lisp machine, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Lisp_machine
[32] Article on High-level language computer architectures, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/High-level_language_computer_architecture
[33] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. 1988. Performance effects of architectural complexity in the Intel 432. ACM Trans. Comput. Syst. 6, 3 (August 1988), 296-339. DOI=http://dx.doi.org/10.1145/45059.214411
[34] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir and M.J. Irwin Use of Local Memory for Efficient Java Execution p. 0468, IEEE International Conference on Computer Design (ICCD01), 2001
[35] S. S. Shekhar, Y. N. Srikant "Object Cache: An Energy Efficient Cache Architecture
[36] Vijaykrishnan, N.; Ranganathan, N., "Supporting object accesses in a Java processor," in Computers and Digital Techniques, IEEE Proceedings, vol.147, no.6, pp.435-443, Nov 2000 doi: 10.1049/ip-cdt:20000787
[37] Lebsack, C.S.; Chang, J.M., "Using scratchpad to exploit object locality in Java," in Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on , vol., no., pp.381-386, 2-5 Oct. 2005 doi: 10.1109/ICCD.2005.111
[38] Srisa-an, W.; Lo, C.-T.D.; Chang, J.-M., "Active memory processor: a hardware garbage collector for real-time Java embedded devices," in Mobile Computing, IEEE Transactions on , vol.2, no.2, pp.89-101, Apr-Jun 2003 doi: 10.1109/TMC.2003.1217230
[39] Schoeberl, M., "A Time-Predictable Object Cache," in Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on , vol., no., pp.99-105, 28-31 March 2011 doi: 10.1109/ISORC.2011.22
[40] Chang, J.M.; Gehringer, E.F., "Evaluation of an object-caching coprocessor design for object-oriented systems," in Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on , vol., no., pp.132-139, 3-6 Oct 1993 doi: 10.1109/ICCD.1993.393393
[41] Esmaeilzadeh, H.; Blem, E.; St. Amant, R.; Sankaralingam, K.; Burger, D., "Dark Silicon and the End of Multicore Scaling," in Micro, IEEE , vol.32, no.3, pp.122-134, May-June 2012 doi: 10.1109/MM.2012.17
[42] Wolczko, M.; Williams, I., "The influence of the object-oriented language model on a supporting architecture," in System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on , vol.i, no., pp.182-191 vol.1, 5-8 Jan 1993 doi: 10.1109/HICSS.1993.270746
[43] Article on Object destruction, available from: (Accessed 7-Dec-2015) https://en.wikipedia.org/wiki/Object_lifetime#Object_destruction
[44] Description of the "The native and Java heaps" from IBM Kwoledge Center http://www-01.ibm.com/support/knowledgecenter/SSYKE2_5.0.0/com.ibm.java.doc.diagnostics.50/diag/problem_determination/aix_mem_heaps.html
[45] Image available from: http://i.stack.imgur.com/ZzVu5.png
[46]http://www.slideshare.net/slideshow/embed_code/27597947?startSlide=7
[47] The Dacapo BenchMark Suite available at: http://dacapobench.org/

[48] Image available from: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
[49] Tutorial on the implementation of smart pointers, available from: http://www.codeproject.com/Articles/15351/Implementing-a-simple-smart-pointer-in-c
[50] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: have your scratchpad and cache it too. In Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15). ACM, New York, NY, USA, 707-719. DOI=http://dx.doi.org/10.1145/2749469.2750374

List of definitions

.. ...

Appendix

A

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	62259	3778274	12520	1757900
4-4-64	43252	3797281	8281	1762139
8-8-128	28646	3811887	4936	1765484
16-16-128	19930	3820603	2722	1767698

Table A.1: Level1 Cache Traffic with LOS – Just Alloc Without ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	19362	55417	17466	2088646
4-4-64	17068	34465	17425	1786479
8-8-128	15442	18140	17339	1534149
16-16-128	15408	7244	17227	1281781

Table A.2: Level2 Cache Traffic with LOS – Just Alloc Without ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocations	Field Writes	Purge Count
2-2-32	36828	1433728	10000	880000	9968
4-4-64	34493	1370752	10000	880000	9932
8-8-128	32781	1285568	10000	880000	9872
16-16-128	32635	1274624	10000	880000	9744

Table A.3: LOS and Main memory and traffic – Just Alloc Without ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	97066	6170969	30952	4784860
8-64	37194	6230841	25054	4790758
16-128	24858	6243177	21568	4794244
32-128	20105	6247930	20526	4795286

Table A.4: Level1 Cache Traffic – Just Alloc Without ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	40631	87387	0	3221056	40631	1524864
8-64	37082	25166	0	2055296	37082	1423168
16-128	34757	11669	0	1650112	34757	1323648
32-128	34757	5874	0	1524864	34757	1323648

Table A.5: Level2 Cache and Main Memory Traffic – Just Alloc Without ROI

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	953126	37742943	141423	19915764
4-4-64	684231	37965667	70279	19982205
8-8-128	299072	38326405	6792	20042889
16-16-128	245109	38364447	4819	20042298

Table A.6: Level1 Cache Traffic with LOS – SmartDB Without ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	144728	949821	133006	53557990
4-4-64	34342	720168	100901	31413441
8-8-128	25805	280059	87583	16467162
16-16-128	27950	221978	84355	11646157

Table A.7: Level2 Cache Traffic with LOS – SmartDB Without ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocs	LOS Deallocs	Field Reads	Field Writes	Purge Count
2-2-32	277734	9159296	140003	3	1137003	2685865	139970
4-4-64	135243	6843520	140003	3	1183174	2690568	139938
8-8-128	113388	6060864	140003	3	1207595	2693371	139874
16-16-128	112305	6021568	140003	3	1223516	2695935	139746

Table A.8: LOS and Main memory and traffic – SmartDB Without ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	1126986	71758335	425521	52563863
8-64	572349	72312972	202221	52787163
16-128	458286	72427035	187664	52801720
32-128	218869	72666452	140178	52849206

Table A.9: Level1 Cache Traffic – SmartDB Without ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	359047	1193460	0	38195712	359047	11030208
8-64	211253	563317	0	20578304	211253	9061504
16-128	198590	447360	0	18469120	198590	8715840
32-128	198590	160457	0	11030208	198590	8715840

Table A.10: Level2 Cache and Main Memory Traffic – SmartDB Without ROI

L1/LOS/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
2-2-32	58742	1481372	14200	809828
4-4-64	36049	1504065	5884	818144
8-8-128	25087	1515027	3795	820233
16-16-128	17359	1522755	2733	821295

Table A.11: Level1 Cache Traffic – Stream Without ROI

L1/LOS/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit
2-2-32	16241	56701	0	1158016
4-4-64	13503	28430	0	608512
8-8-128	9259	19623	0	453248
16-16-128	9259	10833	0	289920

Table A.12: Level2 Cache Traffic with LOS – Stream Without ROI

L1/LOS/L2 (KB)	MM Read	MM Write	LOS Allocs	LOS Deallocs	Field Reads	Field Writes	Purge Count
2-2-32	16241	239872	29	4	116	169	0
4-4-64	13503	192384	29	4	116	169	0
8-8-128	9259	85248	29	4	116	169	0
16-16-128	9259	85248	29	4	116	169	0

Table A.13: LOS and Main memory and traffic – Stream Without ROI

L1/L2 (KB)	L1 Read Miss	L1 Read Hit	L1 Write Miss	L1 Write hit
4-32	37883	1573697	5992	827861
8-64	26604	1584976	3895	829958
16-128	18708	1592872	2807	831046
32-128	15373	1596207	2253	831600

Table A.14: Level1 Cache Traffic – Stream Without ROI

L1/L2 (KB)	L2 Read Miss	L2 Read Hit	L2 Write Miss	L2 Write hit	MM Read	MM Write
4-32	17626	26249	0	618944	17626	250496
8-64	14878	15621	0	463104	14878	208512
16-128	10445	11070	0	297536	10445	99392
32-128	10445	7181	0	250496	10445	99392

Table A.15: Level2 Cache and Main Memory Traffic – Stream Without ROI