Cache Line Reservation: Exploring a Scheme for Cache-Friendly Object Allocation

Ivan Bilicki

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

September 2009

A thesis submitted to McGill University in partial fulfilment of the requirements for the degree of Master of Engineering.

2009/09/24

Abstract

This thesis presents a novel idea for object allocation, cache line reservation (CLR), whose goal is to reduce data cache misses. Certain objects are allocated from "reserved" cache lines, so that they do not evict other objects that will be needed later. We discuss what kinds of allocations can benefit from CLR, as well as sources of overhead. Prototypes using CLR were implemented in the IBM®J9 Java™virtual machine (JVM) and its Testarossa just-in-time (JIT) compiler. A performance study of our prototypes was conducted using various benchmarks. CLR can offer a benefit in specialized microbenchmarks when allocating long-lived objects that are accessed in infrequent bursts. In other benchmarks such as SPECjbb2005 and SPECjvm2008, we show that CLR can reduce cache misses when allocating a large number of short-lived objects, but not provide a performance improvement due to the introduced overhead. We measure and quantify this overhead in the current implementation and suggest areas for future development. CLR is not limited to Java applications, so other static and dynamic compilers could benefit from it in the future.

Résumé

Cette thèse présente une nouvelle idée pour l'attribution d'objet, réservation de ligne d'antémémoire (RLA), dont le but est de réduire les échecs d'accès à l'antémémoire. Certains objets sont alloués à partir d'une antémémoire de réserve, de manière à éviter l'éviction d'autres objets ultérieurement requis. Nous discutons les types d'allocation qui peuvent bénéficier de RLA, ainsi que les sources de coûts indirects. Les prototypes que nous avons développé qui font usage de RLA furent créés utilisant la machine virtuelle IBM®J9 Java™et son compilateur juste-a-temps Testarossa. Une étude de performance de nos prototypes fut conduite avec plusieurs tests de performance. RLA peut offrir un bénéfice pour des microtests de performance spécialisés dans les cas où des objets de longue vie sont lus en rafales infréquentes. Dans d'autres tests que SPECjbb2005 et SPECjvm2008, nous démontrons que RLA peut réduire les échecs d'accès a l'antémémoire dans les cas d'allocation d'un grand nombre d'objets de court temps de vie, mais n'offre pas d'amélioration de la performance vue l'introduction de coûts indirects. Nous mesurons et quantifions ces coûts dans notre implémentation courante et suggérons des domaines de développement futurs. RLA n'est pas limité aux applications Java, ce qui permet à d'autres compilateurs statiques comme dynamiques d'en tirer profit dans l'avenir.

Acknowledgements

I would like to thank the following people that made this thesis possible. My supervisor, Željko Žilić for his advice and guidance throughout my Masters program, and for giving me the independence to work on research that I find interesting. Managers at IBM, Marcel Mitran and Emilia Tung, for allowing me to use IBM resources for my research. Vijay Sundaresan, Nikola Grčevski and Daryl Maier for providing me with the initial CLR idea, and offering regular advice about the technical aspects of the project, and being always available for questions. Yan Luo, for answering numerous questions about the workings of the J9 JVM. Ankit Asthana, Joran Siu, Ted Herman, Andrew Mehes, Bryan Chan (and others) for helping me understand the Testarossa JIT better. My McGill colleagues and lab mates, Nathaniel Azuelos for translating the abstract to French and Bojan Mihajlović for offering advice on writing this thesis. My friend Goran Obradović, for helping out with the testing the proof-of-concept program. I am also grateful to Natural Sciences and Engineering Research Council for providing me with financial assistance.

On a personal level, I would like to thank Jelena Petronjev, for her long online chats that made sitting in front of a computer for hours enjoyable, Nevena Francetić, for giving me encouragement (and food) at various stages, my cousin, Maja Božicki, for being present in my life and giving me numerous advice, and everyone in my "extended family" (you know who you are). Finally, I would like to thank my parents. They are the reason that I exist. In particular, I will always value my mother's dedication and ambition that helped me get where I am today.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

CLR	Cache Line Reservation
JIT	Just-in-time Compiler
(J)VM	(Java) Virtual Machine
TLH	Thread Local Heap
GC	Garbage Collection
CPU	Central Processing Unit
L1 cache	Level 1 CPU cache memory
RAM	Random Access (Main) memory
chunk	cache size/associativity

Chapter 1

Introduction

The Java programming language offers the flexibility required for implementing large and complex programs, and the object-oriented nature of the language allows programmers to abstract functionality into classes and packages. It is common in programming models such as this to instantiate an object and invoke one or more methods on the object in order to perform even a relatively simple computational task. Thus, in order to complete complex transactions, modern server/middleware applications typically end up creating a very large number of objects, many of which are only used for a short duration. Studies have shown that a significant number of objects die young [1], or even instantly [2]; these are referred to as *short-lived* and *zero-lifetime* objects. With all these objects being allocated, efficient memory management is essential.

Locality of reference states that computer programs usually repeatedly access data related either spatially or temporally. If the program accesses a certain memory location M, it can be expected that it would access some other memory location close to memory location M soon (*spacial* locality). There is usually also a strong likelihood that if a certain memory location is accessed once, it might be accessed again several times in a relatively short duration (*temporal* locality). A good overview of caches, locality, and other concepts presented in this thesis is provided in [3].

A CPU cache is used by the processor to reduce the average time to access main memory (RAM). The cache is a smaller, faster memory that stores copies of the data from the most frequently used main memory locations. When the processor needs to read or write a location in main memory, it first checks whether that memory location is in the cache.

This is accomplished by comparing the address of the memory location to all the locations in the cache that might contain that address. If the processor finds that the memory location is in the cache, this is referred to as a *cache hit*; and if it does not find it in the cache, it is called a *cache miss*. In the case of a cache hit, the processor immediately reads or writes the data in the *cache line*. If a program behaves in accordance with the locality of reference principle, most memory accesses would be to cached memory locations, and the average latency of memory accesses would be closer to the cache latency than to the latency of main memory.

Addresses in both kinds of memory (main and cache) can be considered to be divided into cache lines. A cache line refers to a contiguous range of addresses where the size of this range varies on different computer architectures (e.g. from 8 bytes to 512 bytes). The size of the cache line is generally larger than the size of the usual access requested by a CPU instruction, which ranges from 1 to 64 bytes. When a memory access is to a location that is not found in the cache, the entire cache line that the location belongs to is read from main memory and brought to the cache memory. The prior data that was in the cache line is evicted from the cache, so future accesses to that data would have to access main memory.

The cache line replacement policy decides where in the cache a copy of a particular entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called *fully associative*. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is *direct mapped*. Many caches implement a compromise, and are described as *set associative*. So, N-way set associative means that any particular location in main memory can be cached in either of N entries in the cache memory. The simplest and most commonly used scheme to decide the mapping of a memory location to cache location(s) is to use the least significant bits of the memory location's address as the index for the cache memory, and to have N entries for each cache location.

The CPU of a modern computer typically caches at least three kinds of information: instructions, data, and physical-to-virtual address translations. In this thesis, we are concerned only with data caching. Java objects and arrays are allocated in the region of (RAM) memory called the heap. When these objects are created or accessed, load and store instructions reference memory addresses where they are located, and these addresses are brought into the data cache. A load or a store instruction is also called a data access.

A *data access* will produce either a cache hit or a cache miss.

In programs that create a large number of objects (working set), performance can be highly dependent on the cost of accessing memory. Modern Java Virtual Machines (JVMs) employ sophisticated memory allocation and management techniques to increase data locality by laying out objects in memory such that cache misses are reduced (i.e., data being accessed is available in cache memory most of the time). Memory allocation is usually performed by the native code generated on the fly by JIT compilers, whereas memory management is handled by the garbage collector (GC). The GC is a form of automatic memory management where the programmer is responsible for indicating when the objects that require memory on the heap are created, but not for freeing up that memory. When heap memory becomes low, the GC determines which objects are unreachable (and hence dead) and reclaims their memory.

This thesis proposes a novel object memory allocation scheme, *cache line reservation* (CLR)[1], which ensures that a selected allocation is performed at a memory location chosen such that this location would be mapped to a specific cache line. This means that all of the selected allocations map only to a certain portion of the cache "reserved" for those allocations. The criteria for selecting allocations as well as the amount of cache memory to reserve for those allocations could vary (especially depending on the architecture), and we discuss some of them. If the selected allocations are objects that are unlikely to be referenced within a short duration of each other, then it is likely that there would have been a cache miss when these objects are accessed, regardless of the allocation scheme. Therefore, we can improve cache utilization for other (unselected) objects by selecting and allocating these objects such that when the expected cache miss occurs, they evict only other selected objects from the cache.

Part of the work presented in this thesis has been used for [4], which is to be published in Centre for Advanced Studies Conference (CASCON) 2009.

The general idea of CLR is shown to be possible with a simple C proof-of-concept program. A prototype using CLR has been developed using the IBM J9 JVM and Testarossa JIT compiler. We present results on custom benchmarks as well as SPECjbb2005 and SPECjvm2008 benchmarks. The thesis concludes with a discussion of the limitations of the current implementation and ideas for future development.

[1] Patent pending (IBM®Canada Ltd)

1.1 Java and Object Locality

Java is an Object Oriented programming language. It uses the object-oriented paradigm that uses *objects* to group logical constructs together. It allows us to code in discrete units of programming logic, which make the code more readable and structured. Object-oriented languages also utilize other concepts such as polymorphism, encapsulation and inheritance. An "object" is an abstract concept used as a tool for programming. An object can have its own data in in it (whether it is primitive data types or other objects) as well as its own functions (methods). Upon defining an object, we create *instances* of them. In addition, there is a certain hierarchy associated with objects, so they can have parent and child objects. An example of object-oriented programming would be having an object representing a point in a coordinate system, and another object representing a rectangle. Examples of these implemented in Java are given in Listings 1.1 and 1.2.

Listing 1.1 Simple Point Class in Java

```
public class Point
{
    private int valueX = 0;
    private int valueY = 0;

    public Point(int x, int y)
    {
       valueX = x;
       valueY = y;
    }
    public int setX(int x)
    {
       valueX = x;
    }
    public int setY(int y)
    {
       valueY = y;
    }
    public int getX(int x, int y)
    {
       return valueX;
    }
    public int getY(int x, int y)
```

```
    {
        return valueY;
    }
}
```

The `Point` object (in Java, object definitions are called *classes*) is nothing more than a container for two `int` primitive data types. It has *getter* and *setter* methods, `setX` and `getX` for `valueX` and `setY` and `getY` for `valueY`. These are typical in objects. Often, they are called to perform simple tasks. For example, if we wanted to increase the X coordinate of point A by 10, we can achieve that with the following code: `A.setX(A.getX()+10);`. For this simple task, we had to make a call to a function, and then update its `valueX` field (class variable). If we now have another object such as the Rectangle object shown in Listing 1.2, that uses the Point object, we have even more calls. If we wanted to "move" an instance of a Rectangle object by calling the `moveTo` method, we would have to call `moveTo`, and then `setX` and `setY`. Every time a call is made, we have to push the arguments onto the stack and make a jump, and then pop the results off the stack (or adjust the stack pointer).

If the method calls are *virtual*, this can introduce even more overhead. *Virtual methods* occur when it is ambiguous which method to call in a hierarchy tree, because of overloading (another Object Oriented concept). For example, let us say that we have a parent class *Cat* and a child class *Lion*. We also have a method called `setFurColour` in both classes. If we now call the `setFurColour` method on a Cat pointer, we do not know before run-time which `setFurColour` method to call: the generic one in the Cat class, or Lion's own implementation. This will depend on the type of object that we encounter. What ends up happening is that a *virtual function table* is set up for each class. It contains a table of pointers where all the methods for that class are located in memory. The location of this virtual function table can be stored in a hidden static field in the class definition when the JVM loads it. When a call to `walk` is compiled, we have to go to the class definition (whether it is a Cat or a Lion), and look up the location of its virtual function table. If the size of the class instance occupies more than one cache line (more accurately if the space between the class definition field and the field we want to access using the getter or setter method is bigger than one cache line), then this introduces more cache pressure, as we have to bring in more than one cache line to change only one field.

Listing 1.2 Simple Rectangle Class in Java

```
public class Rectangle
{
    private int width = 0;
    private int height = 0;
    private Point origin;

    public Rectangle(Point p, int w, int h)
    {
       origin = p;
       width = w;
       height = h;
    }
    public void moveTo(int x, int y)
    {
       origin.setX(x);
       origin.setY(y);
    }
    public int area()
    {
       return width * height;
    }
}
```

Why one might then use getter and setter methods? The motivation is *modularity* and *data hiding*. We could have used a Point class with two public fields, `public int valueX` and `public int valueY`. Then, we could edit their value using `nameOfInstance.valueX = newValue;` and `nameOfInstance.valueY = newValue;`. However, if we wanted to change the internal implementation of the Point class in the future, we would have to change all calls to its fields. It is much easier to make the outside world not know about the implementation of the object, and just have public methods that modify what is on the inside.

These issues are present in all object oriented languages. This is why object locality is important in general. In addition, the Java programming language makes things even more difficult. Java has a lot of *immutable* objects. Immutable objects are objects that cannot be changed after construction. Examples of these classes are String, Integer, Long, Float and Double. What ends up happening is that the JVM allocates a lot

of short-lived, immutable objects. Even a simple "Hello World" program exposes this: `System.out.println(‘‘Hello World!’’);`. The String containing the characters "Hello World" will create a String object that will die right after creation. By *dying*, we mean that it will never be referenced again. Nothing will contain its reference. When GC occurs, the memory used up by this String will be reclaimed. Why this could become a problem is made obvious in Listing 1.3. We are simply printing out the value of an integer primitive data type 100 times. This will create 100 different instances of the String object, which will die instantly. More practically, each time we try to "modify" any of these immutable objects, we are really killing the old instance and creating a new instance (e.g. converting a string to lower case by calling the `toLowerCase` method). However, the JVM does not know that the space occupied by them is available until after the GC.

Listing 1.3 Creating Many Zero-lifetime Strings

```
public static void main(String args[])
{
   for(int i = 0; i<100; i++}
      System.out.println("Value␣of␣i:␣" + i);
}
```

To conclude this section, object oriented languages use objects as containers for storing data. This increases the memory space requirement for this data than would be needed if primitive data types were used (or structs in C). In addition, getter and setter methods can increase the overhead of updating this data due to calls and virtual functions. Lastly, Java in particular has many immutable objects that die young and increase memory usage. This is why it is important to develop techniques to minimize memory and cache pressure.

1.2 CPU Caches

The central processing unit (CPU) resides on a chip, and performs all the machine instructions that have been compiled by compilers from programs written in high-level languages. Each CPU has its own *instruction set* that it can perform. These instructions fall into three categories:

- Arithmetic instructions
- Memory (data) accesses

- Control instructions (jumps and branches)

Arithmetic instructions operate on registers, and use the arithmetic logic unit (ALU) in the CPU. For example, the contents of two registers could be added and stored in a third register. Registers are small (typically 64 bits for a 64-bit architecture), fast storage spaces. For example, the AMD Opteron processors have 22 registers. Memory access instructions load data from main memory (RAM) into registers, or they store data from registers to main memory (RAM). As arguments, they take a memory address and a register number. Control instructions change the flow of the program (i.e. which instruction is executed next), and will change the flow unconditionally (jump instructions) or based on some condition (branch instructions).

The problem with memory access instructions is that they are several orders of magnitude slower than other instructions. This is because the latency of RAM memory is much bigger than access time of registers (because registers are close to the CPU). The reason for this is that RAM memory has a large relative storage capacity (on the order of gigabytes), and there is always a compromise between the size of memory and how close it can be to the CPU. Cache memory is located close to the CPU, and has a capacity on the order of hundreds of kilobytes. It is used as a fast temporary storage buffer for data that is needed by memory access instructions. The principle of *temporal data locality* states if a memory address A has been accessed, it will be needed again sooner rather than later. The principle of *spatial data locality* states that if a memory address A has been accessed, the next memory address that will be needed, B will be located closer to A rather than further away. Caches are designed to exploit these principles.

Cache memory size is usually in the range from hundreds of kilobytes to a few megabytes. The smallest unit of granularity of a cache is called a *cache line*. It is typically from 1 to 64 bytes. When an address A is required by a data instruction, on first access, it will be loaded from the RAM. A copy of the data will be copied into the cache. The whole cache line containing the address will be brought in from RAM. If the same data is required by a subsequent instruction, this time, it will not be brought back all the way from the RAM, but from the (much faster) cache memory (this is how caches address temporal locality). In addition, if another data reference is encountered close to the previous one, the data will be available in the cache since the whole cache line has been brought in (this is how caches address spatial locality).

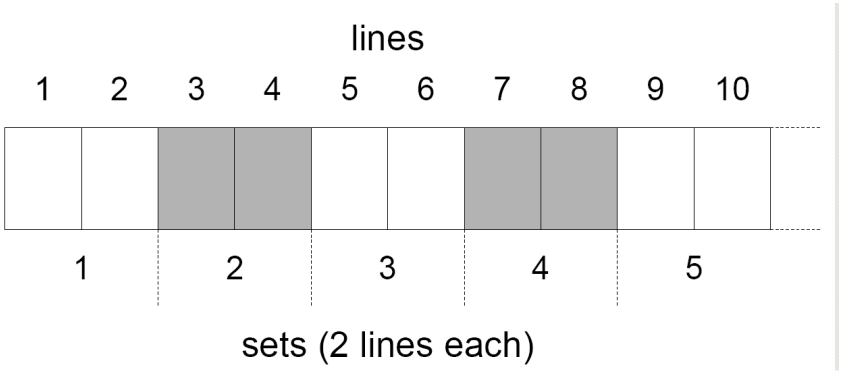

Fig. 1.1 2-way set associative cache)

Apart from being arranged in cache lines, the cache can have *sets* associated with them. Each set has one or more lines in it. If a cache has one line in each set, it is called a *directly mapped* cache. If a cache has 2 lines in each cache, it is called a *2-way set associative* cache (shown in Figure 1.1), and we say that it has an *associativity* of 2. When the CPU encounters an address pointing to data to put in the cache, it first selects the *set* where the data should be placed. Then, within the set, one line is removed (*evicted*) and replaced with the line that contains the data that we need. Please see Figure 1.2 for a representation of how a data address is used to determine where the data is placed in the cache. Going from the least significant bits, the line offset determines where within a line the data will be found. Next is the index, and it determines which set the address maps to. All addresses with the same index will be mapped to the same set. The most significant bits of the address form the tag, which has no say in where the data is being placed in the cache. When a data instruction is encountered, the CPU looks at its operand in the form of an address, specifically the index. Based on the index, it finds the set to which it maps to, and then it compares the tag of the address to each line in that set, to determine if the line containing the data is there already. If it is not, it fetches it from the RAM. The line that is replaced within a set when a cache miss occurs is determined by the cache *line replacement policy*. For example, the *least recently used*(LRU) policy is common. A good overview of all these cache concepts is given in [3].

Cache misses occur when the CPU encounters a memory access instruction, but the data is not found in the cache. There are 3 types of misses:

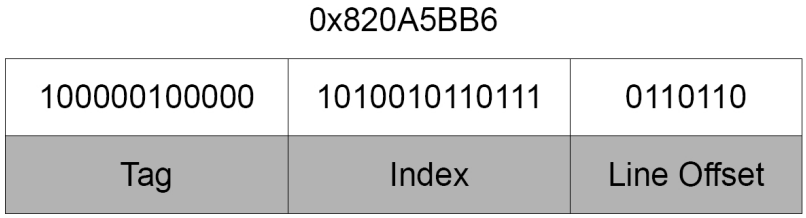

Fig. 1.2 How an address (0x820A5BB6) is decoded for a 2-way set associative cache (512Kb total size, 64-byte line)

- *Compulsory misses* are misses that occur when a data reference is encountered for the first time. These misses are unavoidable, as the data has to be brought back from RAM the first time it is needed.

- *Capacity misses* are misses that occur due to the finite size of the cache. No matter what the cache associativity is, some misses occur due to the cache being smaller than main memory. These misses are also unavoidable given a fixed size of the cache.

- *Conflict misses* are misses that could have been avoided given a certain cache size. These are the misses that we are trying to reduce in this thesis. They come in two flavours: mapping misses and replacement misses. Mapping misses come from the fact that each cache has a certain associativity, and cannot be avoided. Replacement misses could have been avoided if another cache line replacement policy had been chosen, and the data was not evicted as early.

One more concept that is to be introduced is the fact that often, CPUs have another cache memory for code (in addition to data). In addition to needing data from RAM memory to execute data instructions, the CPU also has to fetch from RAM all instructions to be executed. The instruction cache stores these instructions. Of course, a single cache *can* be used for both data and instructions, but this can create an unnecessary bottleneck. In modern CPUs, we do not just have one level of cache memory, but two or even three. For example, the latest AMD Opteron processor at the time of writing (Istanbul), has 6MB of level 3 (L3) cache, 512KB of level 2 (L2) cache and 2*64KB of level 1 (L1) cache (data and instruction).

It is very important to minimize cache misses. The latency of CPU registers is around 0.25ns, CPU cache around 1ns and of RAM memory around 100ns [3]. This means that

the penalty of waiting for data after a cache miss might be hundreds of CPU cycles long. As you see, there is a large difference in the speed of CPU and RAM memory. This is known as the *memory wall*. The rate of improvement of CPU memory speed (on-chip cache and registers) from 1986 to 2000 has been 55%, while only being 10% for off-chip memory (RAM) [5]. As a result, it is becoming increasingly important to use the cache in the most efficient ways possible.

1.3 Dynamic Memory Allocation

Memory management is the process of giving memory to a program when it requests it. Different languages manage memory in different ways. For example, in C++, memory can be allocated statically, automatically, or dynamically. Have a look at Listing 1.4. The string pointed to by `s` is allocated statically at compile time. It will be put in the program binary, and read-only memory. Trying to change the string will result in a segmentation fault. The variable `autoInt` of type `int` will be (most likely) allocated on the *stack* during runtime. A stack is a region of memory used by the program to store information in a LIFO (last in first out) manner. Its main use is to store return addresses of functions, as well as local variables (like `autoInt`). We say that these variables are allocated automatically. For the instance of `myObject` and the array `myArr`, memory has been allocated dynamically. By using the `new` keyword, we told the compiler to allocate, at runtime, enough memory to store one instance of the `myObject` object. Similarly, the function `malloc` returns a pointer to a free chunk of memory (memory reserved for use by the program). It is how this memory is to be deallocated (recycled) that varies the most between programming language implementations. The two main methods are using *manual memory management* or *automatic memory management*.

Listing 1.4 Memory allocation program in C++ (output produced: "Statically allocated string. 5 6 7")

```
#include <malloc.h>
#include <stdio.h>
class myObject
{
  int value;
  public:
  myObject(int a)
```

```
  {
    value = a;
  }
  int getVal()
  {
    return value;
  }
};
void main()
{
  char *s = "Statically␣allocated␣string.";
  int autoInt = 5;
  myObject * id = new myObject(autoInt+1);
  int * myArr = (int*) malloc(6*sizeof(int));
  myArr[3] = 7;
  printf("%s␣%d␣%d␣%d\n", s, autoInt, id->getVal(), myArr[3]);
  delete id;
  free(myArray);
}
```

Object oriented languages create a lot of objects, as mentioned earlier. The reason why we are interested in memory management is because when they are accessed (e.g. if an object field needs to be updated), memory locations that contain the object will be needed and brought into the data cache. Better dynamic memory management can mean a decrease in cache misses. The region of RAM that is used for dynamic allocation is called the *heap*.

1.3.1 Manual Memory Management

Manual memory management is when a programming language requires that the programmer explicitly specifies when to recycle memory. Examples of such languages are C, C++, COBOL and Pascal. In Listing 2.4, this is achieved by using the `delete` keyword and the `free` function. We can think of the `delete` keyword as the opposite of the `new` allocator. When `delete` is called on `id`, its (default) destructor is called, and the memory that was used to store it is freed up (put back into the free memory pool so that it is available for allocation). The `free` function similarly recycles memory and undoes what `malloc` has done.

Manual memory management puts a lot of strain on the programmer because memory bookkeeping is required, and bugs are common. For example, the programmer must not forget to recycle the memory. Otherwise, memory leaks can be produced. Also, if a reference to an object that used to exist survives past memory deallocation of that object, then we can have a dead pointer. Trying to read that memory location in the future, thinking that it contains an instance of the object will result in a segmentation fault (if we are lucky) or us reading nonsense without knowing it.

1.3.2 Automatic Memory Management (Garbage Collection)

Instead of relying on the programmer to know when it is appropriate to free memory, languages that use automatic memory management do it automatically. Examples are Java, Perl and C#. The programmer allocates objects, but does not deallocate them. When the heap fills up, a *garbage collection*(GC) occurs that recycles the memory occupied by any *dead* objects. A *dead* object is an object that cannot be accessed in any way (typically through a reference). After GC, the heap has available space again and the program can continue.

The simplest way to perform a GC would be to start from the *root set* and look for references of objects (this is called a *mark-and-sweep* GC scheme). The root set consists of things like the global and static variables and all addresses that are currently in the registers and the stack. The assumption is that any living object can be reached by starting from the root set. When we encounter a variable in the root set, we have to ask ourselves these questions: is it a reference(pointer)? If so, does it point to an object? If so, we mark the object as live. Then, we have to look at the fields of this object, and repeat the same process recursively. Then, the heap is sequentially walked by the GC, and the linked list of free pointers is rebuilt, because we know which objects are alive and which are dead.

The mark-and-sweep GC scheme is simple to implement, but it has disadvantages. It leads to fragmentation of the heap over time. This can be fixed by compaction. More importantly, it makes the whole program stop during GC. More advanced, *concurrent* GC schemes have been developed, which perform GC in parallel with the running application. Another widespread GC scheme is a *generational* garbage collector. In it, the heap is divided in two parts: the *nursery* and *tenured space*. New objects are allocated to the nursery. When 50% of the nursery fills up, a GC is performed on the nursery only. Objects

that are alive are copied from one half of the nursery to the other. Objects in the nursery that survive enough collections are moved to tenured space. When tenured space fills up, a global GC occurs. The advantages of generational GC are that it improves object locality on the heap (this helps with page faults), less spent time per collection (for each local GC that occurs, the tenured objects are not looked at, and rightly so, because there is a high likelihood that they will not be collected), and less heap fragmentation (copying objects from one side of the tenured space to the other defragments the heap).

Chapter 2

Related Work

Cache misses and caches in general have been thoroughly described in [3]. In addition to what was presented in Chapter 1, it provides cache miss statistics for different cache associativities and block replacement policies. It also describes in more detail how caches operate. On a write, caches can either write the data only to the cache (write back) or both to the cache and to main memory (write through). When a write miss occurs, the line is usually brought into the cache (write allocate), but some caches only modify the line in memory once they have found it (no-write allocate) without bringing the line into the cache. Details of why cache misses occur (compulsory, capacity and conflict misses) have also been explained.

A further understanding of conflict misses has been presented in [6]. In the context of loops, these misses occur when the reuse of data has been disrupted. There can either be self-dependence reuse (one reference accesses the same address in another loop iteration) or group-dependence reuse (multiple references access the same address in different loop iterations). A distinction is also made between temporal reuse (when the exact same address is needed) and spatial reuse (when an address in the same cache line is needed). Conflict (interference) misses make program execution time unpredictable and they are hard to analyze. Some analytical methods for estimating interference cache misses as well as execution time have been presented in [7]. In this chapter, we explore different software and hardware approaches that can be used to reduce cache misses and optimize cache performance in general.

2.1 Hardware Approaches of Improving Cache Performance

2.1.1 Reducing Cache Miss Penalty

Although we have focused more on reducing the actual number of cache misses, cache performance can be improved by reducing the cache miss penalty as well [3] (i.e. how expensive a cache miss is). The first technique that has already been mentioned before is to use *multilevel* caches. On what would be a L1 cache miss, we fetch the data from the L2 cache, not RAM memory. Pulido [8] has performed some analysis on Spice and SPEC'92 benchmarks. They found that on Spice, a 1KB L1 cache results in a cache miss ratio of 19% for the L1 cache. When a L2 cache was introduced, of size 4KB, it resulted in a L2 cache miss ratio of about 10%, which brought the global cache miss rate to 2%. The global cache miss rate is the product of L1 cahce miss rate and L2 cache miss rate. Pulido [8] also shows measurements that indicate that it is benefitial to have a separate cache for data and instructions.

Another technique to reduce the cache miss penalty is to not wait for the whole line to be brought in the cache before the CPU uses it [3]. When a read miss occurs, as soon as the word is found in the RAM, it can be sent straight to the CPU (before it propagates to the L1 cache). This is called the *critical word first* strategy. An alternative is to start reading the words in the line in order, but as soon as the requested word is available in L1 cache, the CPU can continue execution (without waiting for the rest of the line to be brought in). This is known as an *early restart* strategy.

Victim caches can be introduced in CPUs to further reduce the miss penalty [9]. A victim cache is a small unit of cache that is not connected to main memory. When a cache line is evicted from the L1 cache, it is copied to the victim cache (it can have several entries). On a cache miss, when a cache line is not found in the L1 cache, the victim cache is checked in case it contains the line. If it does, we do not have to fetch it from main memory (or the higher level of cache).

A technique widely used in modern CPUs to reduce the stall time after a write is to have a *write buffer*. On a write instruction (for write-through caches), the data to be updated in memory is placed in a buffer. The CPU can then continue execution, and the memory will be updated by the contents of the buffer at its own pace. Of course, now on a read miss, the write buffer has to be checked because it might contain the updated data (as opposed to main memory having it). The efficiency of write buffers can be improved by each entry

having multiple words, so if there is another write instruction close to a word already in the write buffer, it can be combined in the same buffer entry, instead of creating a new one.

2.1.2 Changing Cache Parameters

Cache parameters in terms of total size, line size and associativity can be changed to see how they affect cache misses. For a fixed cache size and a given application, there is an optimum associativity and line size. This is typically investigated by collecting different program traces, and then running them through a cache simulator. As caches have many parameters, the design space is quite large, and it is important to either reduce it, or develop simulators that are fast [10]. A smaller miss ratio does not always mean an improvement in performance. If we increase the line size, although the miss ratio might decrease, the traffic to memory might increase, and the overall performance could be worse [11]. In this thesis, we use execution time as a measure of performance, so this will not be an issue.

Kim [1] has analyzed cache performance on the SPECjvm98 benchmarks in Java using a Jtrace as the tracing tool. They found that garbage collection negatively impacts cache performance and that longer-lived objects are important to cache performance because they determine the size of the working set. In terms of cache parameters, going beyond 2-way associativity improves SPECjvm98 performance only marginally.

Caches in embedded systems can be designed using this technique, where another goal is also low power consumption [12]. Embedded systems typically run very specific applications that do not change. In addition, the hardware is customizable. That is why it makes sense to investigate different cache parameters tailored to the applications. There exist both exact cache models for simulation and approximate ones.

Work presented in [13] uses a forest algorithm that is used to explore a wide design space in terms of cache size, associativity and line size. If we have two caches that have the same associativity and line size, but one is twice the size of the other, then if a cache hit occurs in the smaller cache, we know that it will also occur in the larger cache. Also, if a cache miss occurs for a certain cache size and associativity, then for another cache with the same set size (total size divided by associativity), a cache miss will also occur, if its associativity is larger than that of the original cache. These concepts are used to reuse calculations along the design space and obtain exact cache miss profiles for various cache configurations with only one sweep of a program trace. They also provide a model for

estimating actual execution time and energy consumption.

2.1.3 Non-Standard Cache Topologies

In addition to the general cache model, other different hardware improvements have been tested. For example, *way prediction* improves the time it takes to find the line within a set (in associative caches) [14]. It stores the index of the last used line within the set to predict the next usage. If it predicts right, the latency of finding a line within a set is reduced.

Pseudoassociative caches behave as a direct mapped cache on a hit [15]. But, on a miss, another set is checked for the data (this set can be calculated using a simple hash function). Another similar concept is having a *skewed* cache, which does not have a fixed associativity. For example, in a skewed cache with two lines per set, the first line can be direct mapped, but the second line can be mapped using a hash function as done by Seznec [16]. They show that a 2-way skewed cache can exhibit a similar hit ratio as a 4-way associative cache, for cache sizes in the range between 4KB and 8KB.

There has also been work involving non-trivial line replacement policies. Subramanian [17] has managed to develop a hardware scheme where the line replacement policy can change, based on the application (between LRU, LFU, FIFO and Random). They add two extra hardware elements: a per-set miss history buffer and a parallel tag array for different policies that tracks what sets would be in the cache, if alternate policies were used. When a set needs to be replaced, the policy is chosen based on the history buffer and the corresponding tag array according to the policy that would have a smaller miss rate for that set.

2.1.4 Hardware Prefetching

Prefetching is the act of loading cache memory with lines before they are needed. It is extensively used for instruction caches due to the sequential nature of execution [18]. When only the next (instruction) line is considered for prefetching, this is called *one block lookahead*. A prefetch algorithm can degrade performance due to the increased memory bandwidth and because prefetching might unnecessarily evict another line from the cache that will be needed later. Still, more sophisticated instruction prefetch schemes exist. It is typical to use the branch predictor to predict what code segment comes next. I-cheng [19] found that branch prediction prefetching can achieve higher performance than a standalone

cache 4 times the size, for the benchmarks that they examined.

Data cache prefetching is also possible. Baer [20] uses a reference prediction table and a look-ahead program counter together with a branch prediction table. The look-ahead program counter looks at the instructions ahead of the real program counter, and when it encounters a memory access, the reference prediction table is checked whether it contains the address of the operand from previous executions. If it does, we know which address to prefetch. The branch prediction table is used to predict the value of the look-ahead program counter. It was shown that this scheme is very effective for reducing data cache misses in scientific programs.

Hardware prefetching is advantageous over software prefetching in the sense that it can use the dynamic nature of code during runtime as well as not having any instruction overhead [21]. However, it is difficult to detect complex access patterns in the cases where the compiler cannot detect them in the first place. In addition, hardware prefetching algorithms are wired in hardware, and cannot be tailored to a specific application (or a number of processors).

2.1.5 Scratchpad Memory

Scratchpad memory is a piece of on-chip memory similar to L1 cache, however, the address space of scratchpad memory is different than that of main memory. Therefore, data in scratchpad memory is not present in main memory, and its access time is guaranteed to be on the order of L1 cache. It can be useful for storing, for example, temporary results of calculations or as a CPU stack. The Cell processor uses a similar concept with its SPEs (Synergistic Processing Elements) having their own "local store" memories [22]. The SPEs address their own local memory for instructions and data, and only access system memory through asynchronous DMA operations.

Loop and data transformations can be applied by the compiler if scratchpad memory is available [23, 24]. The compiler inserts instructions so that portions of arrays required by the current computation are fetched from the off-chip memory to scratchpad memory (instead of the cache). As this introduces memory bandwidth overhead, this data needs to be sufficiently reused for a benefit to occur.

Scratchpad memory can also be suitable for allocation of short-lived Java objects [25]. Since they die soon after creation, they will not be written to main memory as they would

be if they were allocated to the L1 cache. This saves the GC time because it does not need to reclaim the memory from the heap (which is in main memory). In addition, the heap is not filled up as quickly. Objects that are longer-lived but are accessed frequently can also benefit from being allocated to scratchpad memory: a process known as pretenuring [26]. By being allocated to the scratchpad, they never evict other lines from the cache, while themselves being always available. Work in [27] performed a static analysis of the objects found in SPECjvm98, and allocated the ones that were referenced the most to scratchpad memory. They found an overall reduction in cache misses and power consumption.

The disadvantage of scratchpad memory is that the size of the scratchpad is needed to be known during compilation, and can be quite variable. As a result, scratchpad memory is mostly used in embedded systems and game consoles (e.g. Sony's Playstation 1 uses a MIPS R3000 CPU with a 1KB of scratchpad memory instead of L1 data cache [28]).

2.2 Software Approaches of Improving Cache Performance

2.2.1 Code Reordering

To improve instruction cache performance, there has been a significant amount of work in the area of code/procedure reordering [29–33]. The idea is to place frequently-used procedures next to each other so that they do not map to the same part of the cache and conflict with each other. Graph colouring algorithms can be used to decide where exactly to place them. Line colouring can also be used to organize arrays in data caches, as opposed to code in instruction caches [34]. Apart from reducing instruction cache misses, the aim of code reordering is also to improve branch prediction [35], which has a direct impact on performance.

Code reordering uses expensive algorithms, and is used in static compilers, but it is also possible to use it dynamically, as shown in [36]. They found that the *code tiling* algorithm produces a performance improvement, unlike the popular Pettis-Hansen algorithm [37]. The main sources of overhead are the generation of the dynamic call graph, and the generation of code layout. Work in [38] uses a compromise: a *mostly correct* control flow graph is generated statically, while the rest of the calculations is finished during runtime. Their aim is to parallelize code, another use of code reordering. The flow graph that is generated statically contains single-entry, multiple exit regions and identifies regions of code for

potential parallelization. This allows for low overhead during runtime.

Huang et al. [39] have managed to implement dynamic code reordering in Jikes RVJ [40], by using three stages: interprocedular method separation (separates hot methods from cold methods), intraprocedular code splitting (separates the hot blocks from cold blocks in hot methods) and code padding (pads the hot blocks to avoid cache overlap when going from one hot block to another).

2.2.2 Software Prefetching

Software prefetching is a technique where the compiler inserts extra prefetch instructions in compiled code [41]. A prefetch instruction brings data to the cache similar to a load or a store, but it does not do anything with it. A prefetch instruction can further be classified into a *register prefetch* and *cache prefetch*, depending on whether it loads the data to a register, or just the cache. A prefetch instruction can be *faulting* or *nonfaulting*. If a *nonfaulting* prefetch instruction is to cause a page fault, it is treated as a no-op.

A simple algorithm is presented in [42] where a prefetch instruction is inserted in inner loops, one iteration ahead. They further go on to recognize that the most overhead is created when data that is already in the cache is prefetched. This can be eliminated by calculating *overflow iterations*, the maximum number of iterations a loop can contain before it cannot hold its data in the cache and cause cache misses. If the number iterations of a loop is smaller than its overflow iteration number, then that prefetch can be eliminated.

Work in [43] uses a special cache, *fetchbuffer*, to help with software prefetching. It stores fetched data and reduce conflicts with the conventional cache. They also prefetch instructions further than one loop iteration ahead, and this distance is based on cache and loop parameters. Prefetching is possible in non-loop environments as well, but the irregular access patterns are difficult to predict [44]. Most prefetching schemes concentrate on memory accesses in loops and arrays, but it is also possible to use prefetching with pointers [45], which is useful for recursive data structures and object-oriented programming in general.

2.2.3 Loop Transformations

In the area of scientific computing, where a lot of computations on large amounts of data are performed, cache performance of specific frequently-executed loops is important. *Loop*

interchange can help with nested loops [46]. If the inner loop has a larger data range than the outer loop, then simply exchanging the nesting of the loops can improve data locality if the arrays do not fit in the cache.

Loop *blocking* or *tiling* is an optimization where data in loops is thought of as being made up of blocks, where one block can fit in one cache [6, 42, 47, 48]. For example, a large matrix multiplication (of size N by N) can be broken down to operate on several submatrices. If we are trying to perform the multiplication of matrices `A` and `B`, and store the result in `C`, then this is done by a series of independent multiplications of elements from `A` and `B` and adding them into elements of `C`. Instead of having an iteration space of N by N, which might not fit into the cache, we can break up the multiplication to smaller chunks, or tiles. This will increase the loop condition overhead, but will improve data locality and reduce cache misses. The challenge is to recognize opportunities for tiling at compile time and determine the correct tile size. Coleman [49] presents an algorithm that chooses an appropriate tile size, based on the problem size, cache size and line size in a direct-mapped cache. Padding can be used to make the data size align to cache line boundaries (or page boundaries [50]). Work in [51] presents some heuristics that can be used for automatic array variable padding.

There are other various techniques, such as *fission* [52], *fusion* [53, 54], and *outer loop unrolling*. What becomes difficult is to select which one of these to use for which specific loop and cache configuration, as the wrong transformation can degrade performance. Wolf et al. [55] developed an algorithm that searches for the optimal set of transformations, while concentrating to minimize the number of introduced loops, and taking care that variables in the loops are register allocatable. *Buffer allocation* is another loop optimization that can be used to reduce the size of a temporary array inside a loop [54].

2.2.4 Improving Object Locality in Memory

It has been recognized that object locality in the heap is important for program execution [56]. The whole heap can be partitioned into multiple sections, where each heap is used for some selected objects. Hirzel [57] uses the connectivity of objects to predict object lifetimes (with the aim of helping us figure out where to put these objects in the heap). Lattner and Adve [58] describe their Automatic Pool Allocation algorithm, whereby they analyze pointer based structures (e.g. a tree structure) and put each of its data instances into a

different pool of the heap, or all of its instances into a single pool, depending on usage. The heap is now contained of many pools, whose memory allocation and deletion needs to be managed by the program, and each static pointer is mapped to the pool it points to. Several programs in C++ have shown a performance improvement.

Object layout can be optimized during garbage collection. The goal is to ensure that objects accessed within a short duration of each other are laid out as close as possible in the heap. Work in [59] presents two interesting tools: `ccmorph` and `ccmalloc`. `ccmorph` uses clustering and colouring algorithms to restructure data structures such as trees to be more cache friendly. `ccmalloc` allocates space on the heap similar to `malloc`, but it takes as an argument a pointer to another structure that is likely to be the parent object accessing to the area of the heap being reserved. `ccmalloc` attempts to put these objects together. A similar process can be used with garbage collection [60]. When the generational garbage collector is copying objects from one part of the nursery to the other, it does it in a way so that objects with high temporal affinity are placed next to each other. For example, Wilson et al. [61] have developed their own algorithm, *hierarchical decomposition*, that can be used by the copying collector to traverse objects (and copy them close to each other). It lead to a significant reduction in page faults.

Although traditionally, garbage collected languages are thought of as having poor cache performance [62], there has been work indicating otherwise [63,64]. Work in [65] presents a GC scheme where objects are classified into two types: *profilic*(frequently allocated/short-lived) and *non-profilic*(infrequently allocated/long-lived). Each of the two types are allocated to a separate area of the heap, and collections are performed independently on the two areas. They found that the the time spent in GC can be reduced by up to 15%, and the heap requirements can be reduced by up to 16%, compared to the generational garbage collection.

2.2.5 Thread Local Heaps

Another way to improve object locality and reduce the overhead of object allocation is to use a *thread-local-heap* (TLH). However, this approach primarily aims at eliminating the need for synchronization at every allocation in the presence of multiple threads. This scheme has been thoroughly described in [66]. In Java, we can have local objects or global objects, from the thread perspective. Local objects can only be accessed by the thread that

created them, whereas global objects can be accessed by multiple threads. We can assign a section of the heap for every thread to allocate objects in. These areas become TLHs. If we know that all objects in a TLH are local, then we can do a local GC on that area of the heap independently from other threads, instead of doing a global collection and stopping all threads. To allocate objects to the TLH, small TLH caches are used. For example, in [66], the TLH cache is at least 384 bytes, whereas the whole TLH is on the order of megabytes. When the cache runs out of space, a new cache is obtained from within the TLH. If there is no space for a new cache, a new TLH can be obtained from the global free list, or a local collection can be initiated. If the object is too large for allocation from the TLH cache, it is allocated from a special large heap area.

Some objects in Java are global by definition, such as Thread and ThreadGroups objects, but it is usually not known *a priori* what objects will be global. All objects start out as being local. Whenever there is an update to a field of an object that is a reference, write barriers are inserted which check whether this reference now points to another object outside the TLH. If it does, the object that it points to is not local any more, and it is marked as being global. Work presented in [66] develops a memory manager that improves the traditional TLH scheme by dynamically monitoring what objects become global based on the allocation site, and instead of putting them in the TLH, another area for global allocations is used. In addition, a static profiling tool was developed which can identify that an object is mostly global, and then the object's bytecode is modified to indicate this. On compilation, the special bytecodes are identified, and these objects are directly allocated into the global areas. The main benefit comes from reduced GC times (by 50%) and GC pauses (by a factor of 3 to 4). Indirectly, it also likely improves data locality, but specific measurements were not performed, as reduction of cache misses was not the main goal. The IBM J9 JVM that we used in this thesis also uses TLHs. In the work presented in the thesis, we modify how objects are allocated from within the TLH *caches* in order to directly reduce CPU cache misses. Unless otherwise stated, the term *TLH* will be analogous to *TLH cache*.

2.2.6 Our approach: Cache Line Reservation

Our scheme differs from prior work in that it uses the mapping/associativity of the cache memory to influence memory allocation with the goal of improving performance by im-

proving locality. To our knowledge, there is no prior work that mentions this approach of object allocation.

However, timing attacks using cache latency are well-known in cryptography, for example to acquire AES (Advanced Encryption Standard) keys [67,68]. As Percival [69] noticed, a Pentium 4 CPU with HyperThreading offers a security vulnerability, because both of the two threads share cache memory. Two processes can communicate in the following way: both processes can allocate arrays that they can access. If sending a 1, the first process can continuously read its whole array (which can be 25% of the cache size) and hence ensure that 25% of the lines are always present in the L1 cache. The second process can read its whole array (which occupies 100% of the cache) and measure the time this takes. When sending a 0, the first process will not be reading its array. Hence, the difference between a 1 and a 0 will be detectable by the second process, and the two processes can communicate. This idea can be extended to allow a spy process to eavesdrop on an AES key.

Chapter 3

Cache Line Reservation

This chapter presents the *cache line reservation* (CLR) concept, which is the main contribution of this thesis. A simple object allocation scheme used conventionally by JIT compilers is based on each thread allocating objects using thread-local heaps (TLH). As mentioned in Section 2.2.5, a TLH is a contiguous chunk of memory of a size controlled by the JIT. A typical implementation of a TLH has two pointers that are of interest to us: an allocation pointer, which we will call *tlhAlloc*, and a pointer marking the end of the TLH pointer, which we will call *tlhTop*. Strictly speaking, the TLH can be a larger area of heap memory and the pointers in question just point to the TLH allocation cache, but from now on, this cache will be referred to as TLH. Even if the TLH approach is not used (e.g. in static compilers), tlhAlloc and tlhTop can be analogous to the allocation pointers in the current free space buffer, indicating its beginning and end. No matter what the implementation is, when an object or an array of primitives need to be allocated, tlhAlloc is incremented by the size of the object, and the object header and fields are inserted in that space. If tlhAlloc is to overshoot tlhTop, a request for a new TLH is made, and the object is allocated from the new TLH.

In the new allocation scheme we present in this thesis, CLR, the idea is to divide each TLH into an unreserved section and one (or many) reserved sections. Selected allocations are performed only in designated reserved sections whereas all other allocations are done in the unreserved section(s). Figure 3.1 demonstrates a high-level overview of CLR in Java. We have a class `myReservedObj`, and a method `myMethod`. In `myMethod`, we are creating a new instance of `myReservedObj`, that we wish to allocate into one of the reserved sections.

When the address for allocation is chosen for where the instance will be allocated, we choose one from RAM memory in such a way so that it maps to a specific cache line (in this case, the first 4 lines in the L1 cache). Which line will be used in L1 cache can be determined by looking at the least significant bits of the allocation address (see Figure 1.2). Therefore, we guarantee that only specific cache lines will be evicted from the cache on allocation, as well as whenever the object is accessed in the future. In the right circumstances, this should decrease cache misses.

3.1 Description of CLR

We now define the CLR concept in more general terms. The size of the unreserved section and each reserved section within a TLH depends on the size and mapping of the cache on the computer that the program is being executed on and the proportion of the cache that is to be reserved for the selected allocations. We begin by defining the following:

`cacheSize`	total size of the cache
`associativity`	the associativity of the L1 cache
`lineSize`	the size of one line in the cache
`reservedSections`	the number of reserved regions in the L1 cache
`chunkSize`	cacheSize/associativity
`tlhSizeMin`	the minimum size of a new TLH allocation buffer
`sectionProportion[reservedSections]`	the proportion of each section
`tlhAlloc[reservedSections+1]`	an array of allocation pointers
`tlhTop[reservedSections+1]`	an array of allocation buffer tops
`tlhHighestAllocIndex`	an index of the highest tlhAlloc entry
`tlhStart`	a pointer that points to the start of the whole TLH
`tlhEnd`	a pointer that points to the end of the whole TLH

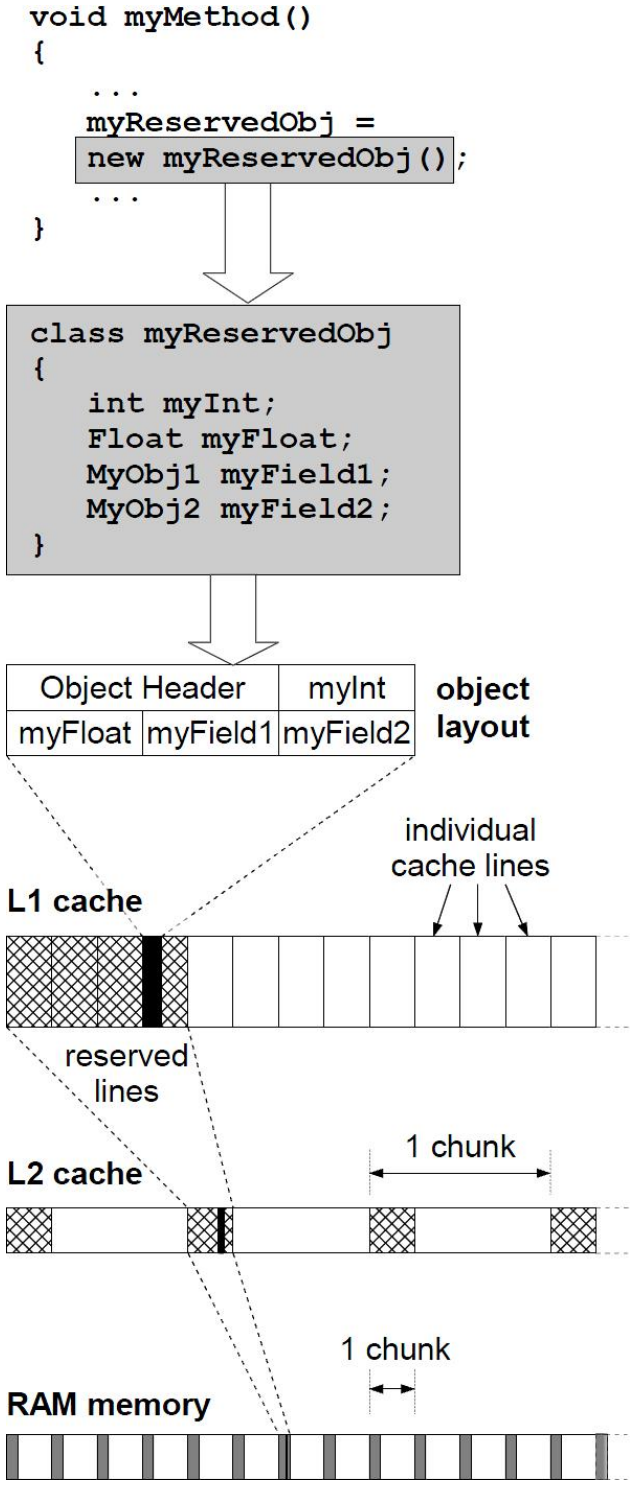

Fig. 3.1 An overview of CLR

Out of these parameters, *cacheSize*, *associativity*, *lineSize* and *chunkSize* are predefined for a given hardware platform. Each TLH should be conceptually viewed as being partitioned into smaller *chunks*. Each such chunk has the property that accessing a given memory location within the chunk would not evict any other memory location within the same chunk from the cache. In other words, each chunk can fit exactly within the cache and different memory locations within the chunk are mapped to different memory locations on the cache. Reservation is done by selecting specific memory locations within chunks such that those memory locations always map to the same cache line(s) on the computer. For caches which have an associativity of 1, the chunk size is equal to the total cache size. As the associativity increases, the chunk size decreases. The maximum number of sections a TLH may be subdivided into is fixed at some small quantity *reservedSections*. This number could change throughout the lifetime of an application. The unreserved section and all the reserved sections form one chunk (see Figure 3.2). The proportion in which these sections are split up between each chunk are stored in the *sectionProportion* array. The elements of this array add up to 1:

```
sectionProportion[0] + sectionProportion[1] + sectionProportion[2] + ...  +
sectionProportion[reservedSections] = 1
```

This array has `reservedSections+1` elements, one more than the number of reserved sections, because location at index 0 will store the data for the unreserved section. For example, if we have 2 reserved sections, `sectionProportions[0]` could be 0.5 (50%), `sectionProportions[1]` could be 0.2(20%) and `sectionProportions[2]` could be 0.3 (30%). The cache lines contained in each reserved section are consecutive, and its proportion could be 0 (if that section is not currently used). The elements in `sectionProportions` array could be changed at runtime. If we define the total number of cache lines contained in one chunk as:

```
numLinesChunk = chunkSize/lineSize
```

we can say that we have `numLines` available to distribute our sections in. The number of consecutive lines (rounded to the nearest integer) in a section `n` is given by:

`linesInSection[n] = sectionProportion[n]*numLinesChunk`

Then, we can calculate the size of each section `n` within one chunk:

`reservedChunkSize[n] = linesInSection[n]*chunkSize`

Of course, the whole TLH is likely to contain more than one chunk, given by:

`(tlhEnd  tlhStart)/chunkSize = numChunksTLH`

The traditional way (without CLR) to allocate objects would be to simply increment the tlhAlloc pointer by the size of the object. Listing 3.1 shows a function for allocating the object without CLR. It returns the pointer to the address of where the beginning of the object is. Then, another function can be called on that pointer to initialize the object header. If the TLH is too small to satisfy the request, a new TLH is obtained. If the object size, `objectSize` is larger than the minimum size of a new TLH, then the object is allocated in another way. These objects are generally too large for CLR allocation anyway. It only makes sense to allocate an object in a reserved area if at least one such object can fit in the reserved section within one chunk (`reservedChunkSize[n]`).

Listing 3.1 Allocating an object in a traditional way (pseudocode)

```
void * startOfObject allocateNoCLR(void * tlhAlloc, void * tlhEnd, int
    objectSize)
   {
   startAllocation:
   if (tlhAlloc + objectSize <= tlhEnd)
      {
      startOfObject = tlhAlloc;
      tlhAlloc = tlhAlloc + objectSize;
      return startOfObject;
      }
   else
      {
      if (objectSize <= tlhSizeMin)
         {
         requestNewTLH();
         goto startAllocation;
```

```
         }
      else
         return allocateFromLargeHeapSpace(objectSize);
      }
   }
```

The allocation incorporating the CLR scheme is more complicated. For each section within a TLH, we must keep track of where the next allocation should occur (represented by `tlhAlloc[n]`). Apart from keeping these running pointers to decide where to do the next allocation, we also need to keep track of the end of each section (represented by `tlhTop[n]`) currently being allocated to. We will assume that within one chunk, the first section is the unreserved section (`n=0`), and that all subsequent sections (`n>0`) are reserved sections. Before allocations begin, each time we get a new TLH, we need to initialize these pointers, based on the beginning and end of the TLH (`tlhStart` and `tlhEnd`). Listing 3.2 shows pseudocode for the initialization. We are assuming that the TLH is bigger than at least one chunk (`numChunksTLH > 1`). But, the TLH does not have to begin on a chunk boundary.

Listing 3.2 Initializing the allocation pointers (pseudocode)

```
void initializePointers(void * tlhStart, void * tlhEnd)
   {
   void * currentPointer;
   int_32t currentSize;
   //go to the beginning of the current chunk
   currentPointer = tlhStart AND (NOT(chunkSize - 1));
   tlhHighestAllocIndex = reservedSections;
   for i = 0 to reservedSections
      {
      if (tlhStart < currentPointer)
         tlhAlloc[i] = currentPointer;
      else
         {
         tlhAlloc[i] = currentPointer + chunkSize
         tlhHighestAllocIndex = i;
         }
      currentSize = ((int) sectionProportion[i]*numLinesChunk) *
         chunkSize
      tlhTop[i] = tlhAlloc[i] + currentSize;
```

```
        currentPointer += currentSize;
        }
    }
```

The initialization function takes `tlhStart` and goes to the beginning of the chunk that it is in. Then, it pretends that that is the beginning of the TLH, and assigns the allocation pointers according to the sizes in the `sectionProportion` array. In the beginning, these allocation pointers will be below `tlhStart`, because we went back to the beginning of the chunk. In that case, `chunkSize` is added to the allocation pointer, to bump it back into the TLH. Similarly, the `tlhTop` array has been initialized. Figure 3.2 illustrates an example TLH and how its pointers look after allocation.

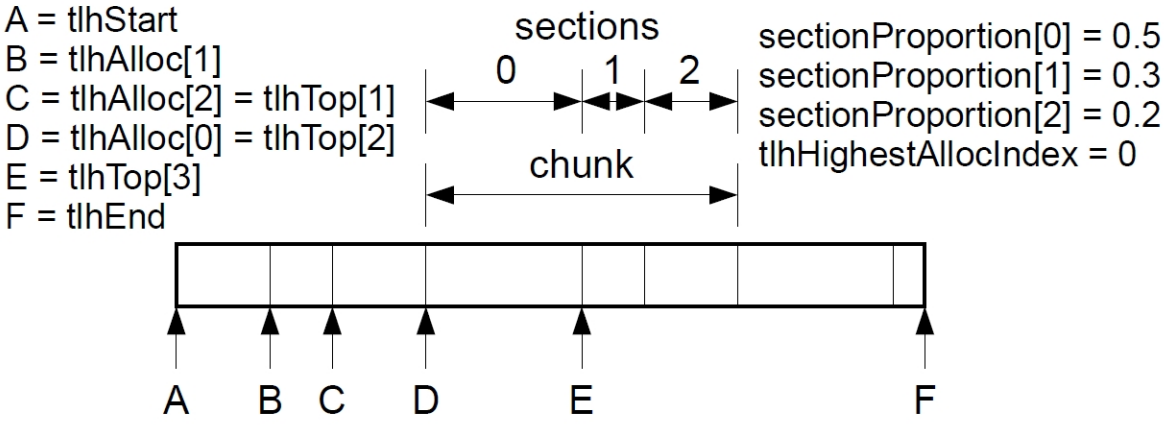

Fig. 3.2 CLR pointer structure and initialization

There might be some space loss between the `tlhStart` pointer and the beginning of the next section. This can be avoided by checking what section `tlhStart` is in and setting that section's allocation pointer to `tlhStart`. The `tlhHighestAllocIndex` is set to be the index of the section whose `tlhAlloc` pointer is the highest. Knowing what entry in the `tlhAlloc` array is the highest will be useful for cache *pollution*. Cache pollution occurs when we cannot allocate the whole object in its designated area (whether it is reserved or non-reserved). This could be because the object is too big, or for other implementation reasons. Now we present the function for allocating non-reserved objects of size `objectSize` in Listing 3.3.

Listing 3.3 Unreserved allocation (pseudocode)

```
void * startOfObject allocateNonReserved(int objectSize)
{
unreservedAllocation:
if (unreservedTlhAlloc + objectSize <= tlhTop[0])
   {
   startOfObject = tlhTop[0];
   tlhTop[0] += objectSize;
   return startOfObject;
   }
if (objectSize < sectionProportion[0]*chunkSize)
   {
   // the object can fit in one section, but there is no space
   if ( (tlhAlloc[0] AND (NOT(chunkSize - 1)) + chunkSize < tlhEnd)
      {
      //jump to new tlhTop[0] pointer in the next chunk
      tlhAlloc[0] = tlhAlloc[0] AND (NOT(chunkSize - 1) + chunkSize;
      tlhTop[0] = tlhAlloc[0] + ((int) sectionProportion[0]*numLinesChunk
         ) * chunkSize;
      //check that we did not overshoot tlhEnd
      if (tlhTop[0] > tlhEnd)
         tlhTop[0] = tlhEnd;
      goto unreservedAllocation;
      }
   else
      {
      //no space in the TLH
      requestNewTLH();
      initializePointers(tlhStart, tlhEnd);
      goto unreservedAllocation;
      }
   }
if (objectSize > tlhSizeMin)
   {
   //object size too big for TLH
   return allocateFromLargeHeapSpace(objectSize);
   }
//object can fit in the TLH, but not in the unreserved section
// allocate with cache pollution over the reserved sections
```

```
if (tlhAlloc[tlhHighestAllocIndex] + objectSize > tlhEnd)
   {
   //There is enough space in the current TLH
   startOfObject = tlhAlloc[tlhHighestAllocIndex];
   tlhAlloc[tlhHighestAllocIndex] += objectSize;
   //Adjust the TLH pointers of all reserved sections that
   //may now be covered by this polluting allocation.
   fixPointers(tlhAlloc[tlhHighestAllocIndex], tlhEnd);
   return startOfObject;
   }
else
   {
   //Get a new TLH and allocate the object
   requestNewTLH();
   startOfObject = tlhAlloc;
   //Initialize the CLR pointers
   fixPointers(startOfObject+objectSize, tlhEnd);
   return startOfObject;
   }
}
```

The `allocateNonReserved` function first looks at whether it can simply increment the pointer for the non-reserved section. This will happen most of the time. However, if the object cannot fit in the current non-reserved section, it tries to update the pointer to the next chunk and then allocate it from there. If there is no more space in the TLH, a new TLH request is made, before trying to allocate the object again. If the object is larger than the minimum size of a new TLH, then the object is allocated from the large heap space. If the object is larger than the non-reserved section size, but smaller than the minimum TLH size, then cache pollution occurs. We cannot just use the `tlhAlloc[0]` pointer to allocate such an object, because the object could end up overlapping with another (reserved) object in the next section. Instead, we allocate it from the section which has the highest `tlhAlloc` pointer, by using `tlhHighestAllocIndex`. This will not interfere with any objects because there are no objects between that pointer and the end of the TLH. However, now when other sections try to update their `tlhAlloc` pointers, they might update it in the middle of this large object. That is why a new function has to be called, `fixPointers`, which is shown in Listing 3.4.

Listing 3.4 Fixing the allocation pointers after cache pollution (pseudocode)

```
void fixPointers(void * fixStart, void * tlhEnd)
   {
   initializePointers(fixStart, tlhEnd);
   for i = 0 to reservedSections
      {
      if (tlhAlloc[i] < tlhEnd)
         tlhAlloc[i] = tlhEnd;
      if (tlhTop[i] < tlhEnd)
         tlhTop[i] = tlhEnd;
      }
   }
```

The `fixPointers` function calls `initializePointers` to initialize the pointers as before, but it pretends that `fixStart` is the beginning of the TLH. As a result, some pointers will overshoot the end of the real TLH, and if that happens, it simply sets those pointers to be `tlhEnd`. No harm is done as on next allocation from these sections, it will be found that there is no space in the TLH, and a new TLH will be requested. Figure 3.3 illustrates what happens when we encounter an object larger than the unreserved section. We have no choice but to pollute the cache, but the TLH pointers are re-initialized so that future allocations can continue as usual.

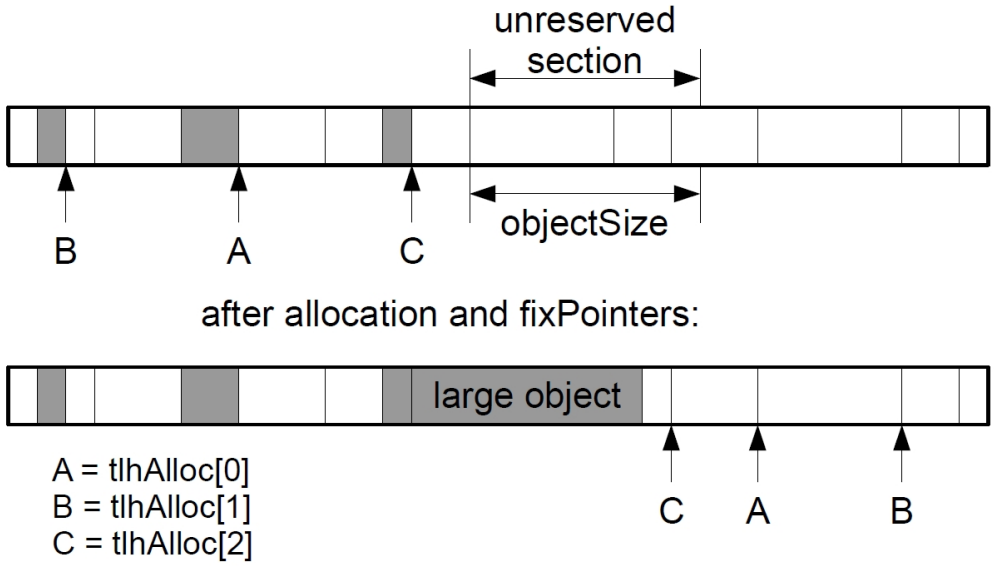

Fig. 3.3 How an object is allocated that cannot fit in the current unreserved section

Listing 3.5 presents pseudocode for a reserved object of size `objectSize`, in section `sectionNum`.

Listing 3.5 Reserved allocation (pseudocode)

```
void * startOfObject allocateReserved(int objectSize, int sectionNum)
{
reservedAllocation:
if (reservedTlhAlloc[sectionIndex] + objectSize <= tlhTop[sectionNum])
   {
   startOfObject = tlhTop[sectionNum];
   tlhTop[sectionNum] += objectSize;
   return startOfObject;
   }
if (objectSize < sectionProportion[sectionNum]*chunkSize)
   {
   // the object can fit in the section, but there is no space
   if ( (tlhAlloc[sectionNum] AND (NOT(chunkSize - 1)) + chunkSize <
      tlhEnd)
      {
      //jump to new tlhTop[sectionNum] pointer in the next chunk
      tlhAlloc[sectionNum] = tlhAlloc[sectionNum] AND (NOT(chunkSize - 1)
          + chunkSize;
      tlhTop[sectionNum] = tlhAlloc[sectionNum] + ((int)
         sectionProportion[sectionNum]*numLinesChunk) * chunkSize;
      //check that we did not overshoot tlhEnd
      if (tlhTop[sectionNum] > tlhEnd)
         tlhTop[sectionNum] = tlhEnd;
      goto reservedAllocation;
      }
   else
      {
      //no space in the TLH
      requestNewTLH();
      initializePointers(tlhStart, tlhEnd);
      goto reservedAllocation;
      }
   }
//object cannot fit in the reserved section
//this should never happen
```

```
return allocateNonReserved(objectSize);
}
```

The `allocateReserved` works similarly as `allocateNonReserved`, only that it is implied that the reserved object size is smaller than the reserved section size. If larger objects need to be allocated in that reserved section, than its corresponding `sectionProportion` entry should be increased. Changing the values during runtime in the `sectionProportion` is an expensive task because of the requirement that the unreserved region is in the beginning of each section (and continuous). If we are to change an entry for section `N` in the array, we will also have to change the entry for index 0 (non-reserved section) by the exact same amount so that all the sections' values add up to 1. But, this will have the effect of moving all the sections below N relative to each chunk, increasing cache pollution.

3.1.1 Choosing the number of cache lines to reserve

Once an allocation site has been chosen for reserved allocation, one parameter that needs to be selected is how many cache lines to reserve for such an allocation. This should be done using a heuristic function that chooses the number based on criteria such as the number of objects that are likely to be allocated (can be estimated by profiling), the size of the objects, the overhead of taking care of TLH pointers as well as the amount of pollution that will be introduced by changing the sections. In any case, the number of cache lines in one reserved section must be enough to hold at least one instance of the object. We might choose to put the reserved object in an existing reserved section (thus sharing it with some other object), or creating a new reserved section. What will happen is that when that method is compiled, the allocation of the selected object from that site will be compiled utilizing CLR.

3.1.2 Alteration to pre-fetch strategy

Some compilers insert prefetch instructions before object allocation in an attempt to decrease cache misses. Without CLR, we can safely just prefetch the data after the `tlhAlloc` pointer. This will bring in the cache lines where the object is to be located before they are needed for object initialization. With CLR, this prefetching strategy will have to be changed. Now, data after `tlhAlloc[n]` should be brought in, unless the pointer has to be adjusted. If that happens, than the new value of `tlhAlloc[n]` should be brought in as

soon as it is known.

3.1.3 Interaction with garbage collection

If the language that CLR is used on uses garbage collection that move objects, then these objects have to be moved in a way so that the CLR benefits are still present after the GC. This means that objects that are inside their appropriate sections have to be moved to those same sections. There are three possible high-level implementations:

- One simple implementation would be to treat chunks as units for garbage collection, instead of objects. A chunk can be collected only if all the objects in that chunk are dead. If not, the chunk as a whole can be moved to another area of the heap, but on a chunk boundary. The advantage of this is that no special knowledge of sections and reservations is needed by the GC, only the chunk size. The disadvantage is that the GC is less efficient, and that any cache pollution that arose from changing the sections will still be there.

- Another way would be to inspect where the objects are located from within the chunks. The GC will have to look at the `sectionProportion` array. Then, the GC can move them to a new area of the heap to the appropriate sections. In effect, the GC will allocate the objects again. This method has a lot of overhead, but it will not be subject to internal fragmentation of the first method.

- The third method could be to mark in the objects themselves what section they belong to. In the object header, one could store a number representing the section to which the object belongs to. This number does not necessarily have to correspond to the index of the `sectionProportion` array, to account for changes of sections during runtime. When the GC finds that it needs to move these objects, all it has to do is check this field in order to see to what section the object can be copied. This approach has medium overhead (no calculation of which section the object is in but the objects now occupy a larger space) but it deals with cache pollution, as the field in the object header is more reliable indicator of which section an object should be in than the object's position within a chunk.

3.1.4 Multiple levels of cache

Today's caches typically contain more than one level. Because CLR works from the lowest level (L1), its benefits will be realized by caches of any levels. But, CLR can be extended to work on higher levels as well. We can either ignore the L1 cache altogether and use the same structure, only with L2 parameters. This will effectively mean that the chunk size has been increased and that the TLH size itself must be larger so that it contains at least one chunk.

An alternative would be to extend the CLR scheme to use two levels. Another set of arrays of pointers will be needed for the L2 cache, and the reserved allocations will have to specify what section (for the L2 chunk) and subsection (for the L1 chunk) they want to allocate to.

3.1.5 Multiple allocation sites per section

If CLR is to dynamically update the `sectionProportion` array, then the way CLR was presented so far, we can only allocate instances of objects from a single allocation site per reserved section. This is because if we want to delete the reserved section (i.e. making its `sectionProportion` entry 0), we can identify the site of allocation that is not required for reserved allocation, and change its allocation code to non-reservation. But, if there is another allocation site that is using the same reserved section, this section does not exist any more. Therefore, in order to reserve objects from multiple allocation sites to one reserved section, we have to make sure that that section never gets deleted.

Alternatively, we could introduce another array, `numAllocationSites`, which be of size equal to `reservedSections`. Each entry is a counter that gets incremented every time code for a reserved allocation is compiled for an allocation site. Then, we know that we cannot make the section proportion 0 unless the corresponding counter in the `numAllocationSites` array is also 0. Conversely, when CLR allocations are cancelled (see Section 3.3.1), the entries in `numAllocationSites` are decremented. If any of them reaches 0, then that section has no objects that are allocated in it, and it might be beneficial to make these sections 0 (but it might not, have a look at Section 3.2.4).

3.2 Criteria for selecting objects

The CLR allocation scheme's goal is to reduce cache misses in correct conditions. To do that, we have to identify what objects are eligible for CLR. Here, we explore all the possible scenarios where CLR could offer a benefit, and how to identify them so that we can (re)compile the allocation code to use CLR.

3.2.1 Frequently Instantiated Types

If there are certain types of objects that are very frequently allocated (per unit of time of execution or relative compared to other objects), then chances are that these objects will be short lived. One extreme would be allocating a lot of zero-lifetime objects using CLR. From the cache perspective, without CLR, they would take up the whole cache, evicting anything that was there before. When longer lived objects are accessed in the future, cache misses occur. With CLR, zero-lifetime objects would only occupy specific cache lines, so that they evict mostly themselves out of the cache (not a problem because they will die and will not be needed in the future anyway). As a result, when other long-lived objects are accessed, there will be more cache hits. If an object type has been identified as being frequently allocated, all we have to do is compile the allocation code for all these objects using CLR.

3.2.2 Frequently Executed Allocation Site

Another way to identify short-lived objects is to look at allocation sites. If we have a section of code that allocates certain objects, and this section gets executed often, then chances are that these objects allocated from *this* site are short-lived, even though the same object types in general could not be. We could find this out by profiling during runtime, or by some sort of static analysis of code. The advantage of this method over just looking at frequently instantiated types in general is that it is more targeted.

3.2.3 Objects that are unlikely to co-exist

There are many objects withing a lifespan of a program that are unlikely to co-exist. If all of the objects that do not co-exist always occupy the same part of the cache, then they

would evict each other out of the cache, instead of other objects that will be needed during their lifetime.

3.2.4 Objects accessed in infrequent bursts

Even if the objects are not short-lived, CLR could help. If a large group of objects is periodically read/written to in isolation, they would evict all the lines from the cache when this happens. But, if they occupy only a specific part of the cache, then they would evict less of the other objects from the cache. For example, we might be searching for an element in a linked list. If each node in the linked list is occupying the same cache line, then every time we search for an element in the linked list, we are only evicting one cache line, instead of the entire cache (if the linked list is large enough).

3.2.5 Mostly Written Objects

Objects that are typically only written to (such as log buffers), would benefit from an exclusive cache line state. It is typically more expensive for a cache nest to be moved from a non-exclusive to exclusive state. If these objects are all located in certain cache lines, then whenever they are written to, the cost of turning the cache nest to an exclusive state has already been paid. This could be more beneficial than if we had to write to other (non-reserved) cache lines.

3.2.6 Objects in Different Threads

Finally, objects allocated by (and potentially only used by) a certain thread can be allocated such that they are mapped to cache lines that are different from those that would be utilized for allocations done by any other thread. This is useful in scenarios where multiple threads running on a different processor are primarily operating on their own thread-local data.

3.3 Limitations of CLR

The CLR allocation scheme, of course, comes at a certain cost. It will depend on how CLR is implemented, as well as how often reserved allocations occur. The idea of CLR it is not limited to any programming language or compiler, but as this thesis focuses on Java, here we explore some sources of overhead from the perspective of a JVM that uses TLHs.

- **Reduced L1 cache capacity for non-reserved objects**
  When reserved sections are allocated, they effectively render these sections useless for non-reserved allocation. That means that the cache capacity has reduced from 100% to the value in `sectionProportion[0]`.

- **More garbage collection**
  If reserved objects are being continuously allocated, in order to satisfy the requirement of them occupying only one part of the cache, there will be a lot of unused space in the TLH when it reaches the top. For example, if the first 25% of the cache is reserved, and there is a section of code that only allocates reserved objects, the TLH will fill up 4 times as quickly. As a result, GC will occur 4 times as often. However, this is only true when *allocating* objects. After they have already been allocated, there is no GC cost in reusing (reading or writing) them.

- **Fragmentation of the heap**
  After a GC, even though the space in the non-reserved areas will be reclaimed, now the free space is heavily fragmented which might make future allocations (reserved or non-reserved) difficult. This effect will be minimized by GC schemes that compact the heap (move objects around to fill up the holes), such as generational GCs. The choice of how the free list of pointers in the memory pool is implemented also affects the degree of this overhead.

- **More expensive allocation**
  Allocation of reserved objects is not as straightforward as incrementing the tlhAlloc pointer in the TLH any more. We have to deal with different sections, initializing and incrementing pointers as well as reading all of these pointers from memory. This increases execution time per allocation, and might even hurt data cache locality (due to more memory reads). Of course, most of the time, the allocations will take the fast path, which is similar to the non-CLR allocation, but it still means going through various conditional branches.

- **Decreased Instruction Cache Locality**
  In addition to more execution time, the size of the compiled methods that use reserved allocation will be bigger. This will hurt locality of the instruction cache.

- **More frequent TLH requests**
  As the individual TLHs fill up more quickly, fresh TLH requests will happen more often, and more time will be spent in the JVM on memory management instead of running useful code. This will also put more strain on the free memory pool management. The memory footprint of the program will appear to be bigger.

- **CLR-aware GC and VM**
  Since not all allocations happen through the JIT compiled code, the allocations from the VM must be compatible with the CLR infrastructure. Similarly, in GC schemes that move objects (compaction, generational GC), the GC might have to be modified so that the reserved objects are moved in a way so that they still map to the reserved regions.

- **Extra instrumentation and compilation time**
  In a fully-automatic JVM with a JIT that supports CLR, extra instrumentation has to be developed that detects what objects to reserve. Then, these methods would have to be recompiled (which takes time). Alternatively, it might be decided that some objects should stop being reserved, which would trigger recompilation again.

The question is whether all these costs are justified by the reduction in cache misses that CLR provides, and under what circumstances. The aim of the CLR implementation presented in the next chapter is to answer this question.

3.3.1 Cancellation policy

The criteria for selecting allocation sites for CLR is described in Section 3.2. However, sometimes, this decision should be reversed, due to the overhead. As mentioned before, each time we are creating a reserved section, we are reducing the cache capacity of non-reserved allocations. If the reserved allocation site stops allocating as many objects, or it disappears (for example, if a method containing it never gets called again), then this reserved area should be reclaimed. In a dynamic CLR scheme, some profiling will be necessary for evaluation of CLR schemes. Here are some examples:

- Tracking the number of allocations for each site
  Each time we perform a reserved allocation, we could increment a counter. This

counter can be checked by a profiler after a certain unit of time to measure the rate of reserved allocations. If this rate falls below a certain threshold, the reserved allocation from that site should be cancelled.

- Tracking the number of new TLH requests
  Similarly, a counter can be incremented each time a TLH is requested. If the rate of TLH requests rises above some threshold, that means that CLR could be impacting performance too much, and that we are spending too much time managing TLH requests and initializing CLR pointers.

- Tracking the number of polluting allocations
  Another easy counter that could be monitored is how often the cache pollution occurs from unreserved allocations. If lots of relatively large objects are being allocated, the introduction of CLR itself might make all of them reserved. Whereas before, they had the whole TLH to their disposal, now the TLH is fragmented, and they might not fit in their unreserved sections.

When an allocation site that used CLR is nominated for cancellation, the allocation code should be recompiled to not use CLR, and the appropriate pointers should be updated (`sectionProportion`, `tlhAlloc` and `tlhEnd` arrays). A convenient time to do this in languages with GC is just before a GC, so that the objects start out in their updated places before execution continues, as well as to clear up as much space in the non-reserved sections as possible.

Chapter 4

Implementation Details

4.1 The TR JIT Compiler and J9 Java Virtual Machine

The basis for our implementation is the proprietary IBM Java Virtual Machine (see Figure 4.1), called *J9*. It uses a highly optimized, high performance JIT (just-in-time) compiler, called *Testarossa*. It has been designed in a modular way, as it can be used with different JVMs, Java class libraries and target architectures. It is currently being developed at the IBM Toronto Software Lab.

4.1.1 The J9 JVM

The J9 JVM [70] is a part of a larger JRE (Java Runtime Environment) that enables Java to run on a specific system (defined by its operating system and CPU architecture). Java high-level code is contained in `.java` files. These files define Java objects, called classes. Javac then compiles the `.java` files to portable Java *bytecode*, with the help from standard class libraries (e.g. `String`, `Thread`, `BigInteger`). Java bytecode is an instruction set for a stack-oriented virtual architecture. This architecture does not exist in hardware and the JVM emulates it. This design decision was made so that Java applications are portable across different systems: all we need is a JVM for that system. Traditionally, the interpreter manipulates the operand stack and interprets the bytecode to run the application. This is very slow. The alternative would be to compile all the methods on startup, but this would greatly increase the startup time. The compromise is dynamic compilation, which is where the JIT compiler is used. It runs faster than an interpreter, and starts up faster

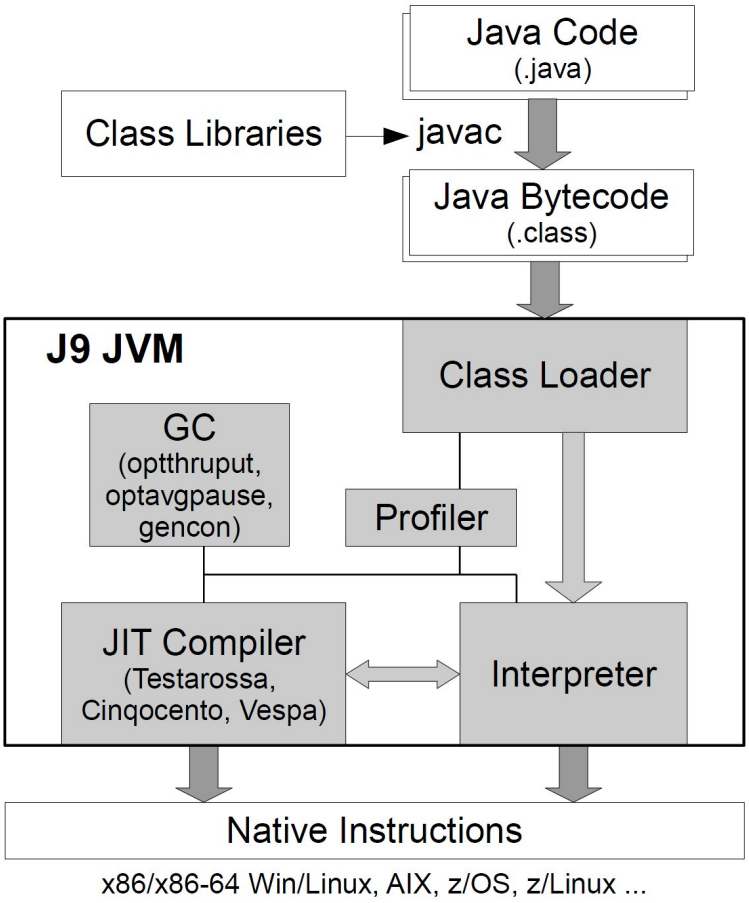

Fig. 4.1 An overview of the J9 JVM and JRE

than pre-compilation.

Figure 4.1 shows the main components of the IBM J9 JVM. The class loader is responsible for loading classes from `.class` files into the JVM as they are needed. The interpreter can then interpret them. There is also a profiler component which samples what is happening in the JVM. The most important task is counting how many times each method gets executed. If this number exceeds a certain threshold, it triggers the compilation of that method by the JIT. Once the method is compiled, the next time it is run, it will run from compiled native code, instead of interpreted. An essential component of any JVM is the garbage collector. IBM's GC has three main schemes: optthruput (the default scheme which is a simple mark-and-sweep scheme with compaction), optavgpause (the scheme designed to reduce the pauses that GC makes during a collection) and gencon (generational GC). The platforms supported include x86 Windows/Linux, x86-64 Windows/Linux, AIX (PowerPC), z/OS and z/Linux. Ahead of time compilation (AOT) is also supported. The JIT has three versions based on how much resources it uses: Testarossa (the full JIT), Cinquecento (used in embedded systems) and Vespa (used in cellphones).

4.1.2 Testarossa JIT

The role of the Testarossa JIT optimizing compiler is to compile frequently used methods when they are needed, (*just in time*).

The JVM we used is IBM's J9 along with the Testarossa (TR) JIT compiler. The first stage is converting the Java bytecodes into an internal *intermediate language*(IL). The Testarossa IL utilizes a tree-like structure. Treetops are connected via a linked list. There is also a control-flow-graph (CFG) that defines how groups of treetops are connected. This is done because it is difficult to perform optimizations on Java bytecode, whereas the Testarossa IL is easily manipulated. The next stage is passing the IL through the optimizer multiple times. The optimizer performs all the standard dynamic compiler optimizations such as value propagation, inlining, escape analysis and tree simplification [71]. The number of optimizations done is dictated by the compilation level of the method (no-opt, cold, warm, hot or scorching). Methods are compiled at an increased level of compilation if they are frequently executed and/or the program spends a lot of time in them (the profiler can identify that). After the IL has been optimized, the IL evaluation stage is next. It converts the IL to machine-specific instructions, assuming an infinite number of registers.

Then, the register assignment stage allocates real registers to symbolic ones and generates spilling instructions where required. The last stage, binary encoding, simply translates the assembly instructions into machine code. The JVM is now ready to run the compiled version of the method the next time it is executed.

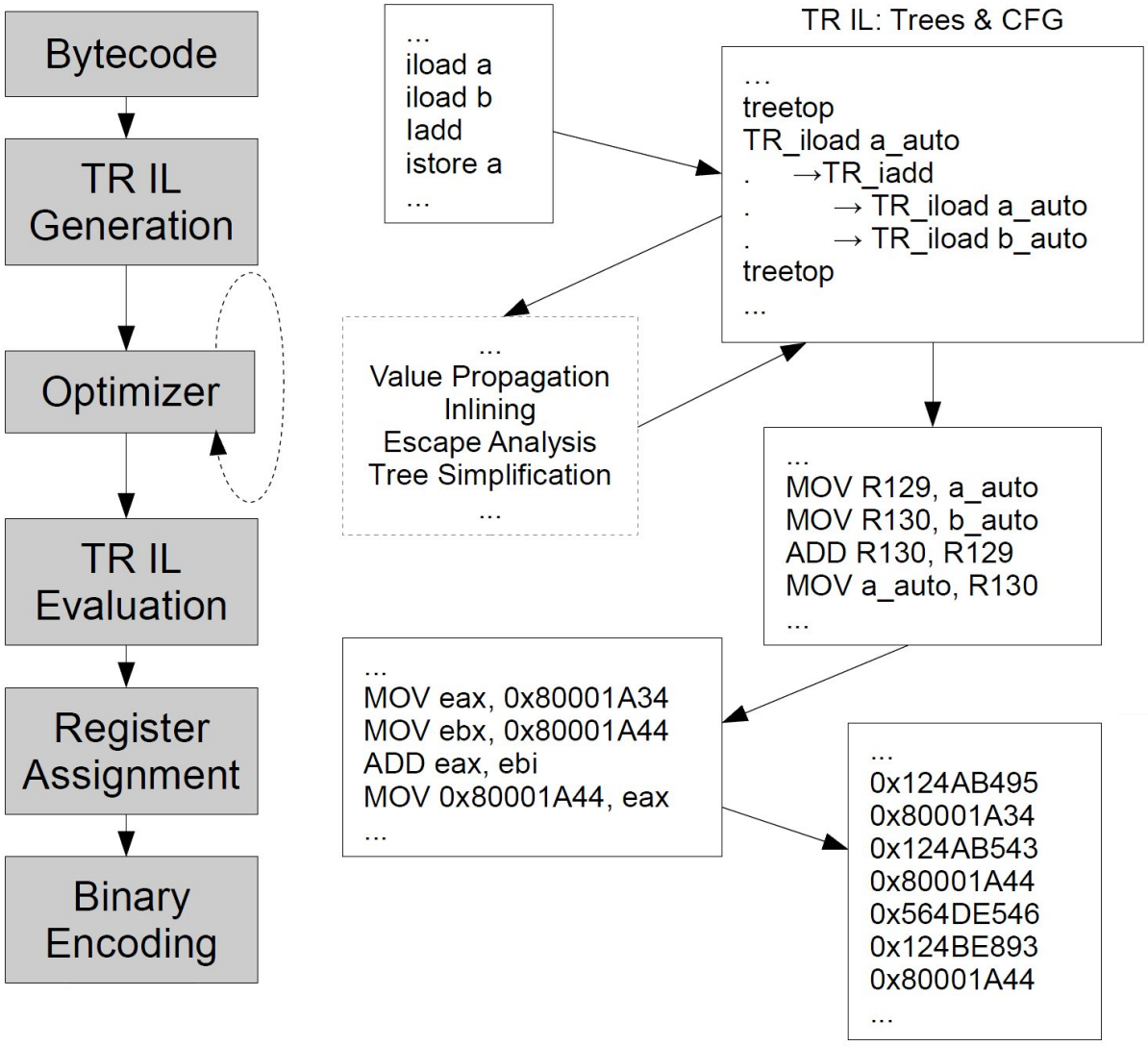

Fig. 4.2 An overview of Testarossa JIT compiler

4.2 Other Tools Used

4.2.1 WinDbg Debugger

The main debugging tool used during development of the CLR prototypes was the WinDbg debugger [72]. It has been used in three main ways. Firstly, it has the ability to load core dumps that are created after the JVM crashes (e.g. due to a segmentation fault). WinDbg can restore the memory and register contents of all threads, the call stack as well as the crashing instruction. Secondly, Windbg can attach itself to a running process and step through the code (either machine instructions, or through the C++ code with the appropriate debugging symbols that are present in the J9 JVM). Thirdly, by inserting a simple illegal instruction (`int 3` on x86), and setting WinDbg as the default postmortem debugger, a breakpoint can be generated in the compiled code that automatically invokes the debugger.

4.2.2 IBM Heap Analyser

In Java, there is no way of knowing where the object is stored. The physical address of Java pointers is not accessible to the programmer (unlike in C/C++, for example). Therefore, in order to know where the objects are located from within the heap, IBM Heap Analyzer [73] was used, a graphical tool for analysis of IBM JVM's heap dumps (those can be triggered by the user). It can create a tree of object hierarchy, and, more importantly, show their locations in memory.

4.2.3 IBM Garbage Collection and Memory Visualizer

The GC and Memory Visualizer is a tool used in the IBM Monitoring and Diagnostic Tools for Java [74]. It can read J9 JVM GC logs and process/summarize them, to give statistics about GC and the heap, such as the number of collections, mean time per collection and the proportion of time spent in GC. It can also draw graphs indicating how the heap size was changing during time.

4.2.4 AMD CodeAnalyst

AMD CodeAnalyst [75] is a program designed to analyze code performance on AMD processors. It uses lightweight profiling to collect data from hardware counters and processes

them after. The metrics interesting to us include the number of memory accesses, L1 cache misses as well as the number of evicted L1 lines. It was used to evaluate the performance of CLR.

4.3 Prototypes Developed

In this section, we discuss details of the CLR prototypes developed. The implementations were done on a development branch of the J9 JVM and Testarossa JIT for Windows from January 2009. The platforms used were x86, x86-64 and x86-64 with compressed references. Compressed references are used in a special version of the J9 JVM and JIT, which tries to reduce the overhead of having larger references and hence larger objects due to the 64-bit addressing mode compared to 32 bits. This is done by reducing the sizes of references used in the JVM.

Java objects are allocated when the following Java bytecodes are encountered: `new`, `newarray` (for array allocations) and `anewarray` (for allocations of arrays of references). The allocations can be invoked through the following paths: from methods that the JIT compiled, from the JVM when a JIT allocation fails (if the TLH gets full or the object is too large for TLH allocation) or from interpreted methods by the JVM. The JIT compiler compiles *frequently* executed methods. Since CLR focuses on frequently allocated and/or frequently used objects, chances are that they will be allocated from the compiled code. Therefore, we are interested in the JIT path, and change the compiled code for these bytecodes in order to incorporate CLR. Without CLR, the JVM allocates objects as explained in Listing 3.1, using the `tlhAlloc` and `tlhTop` pointers.

4.3.1 Prototype 1 - "Weak" CLR Reservation

In our first prototype, we allocate all objects so that their beginning is in a specific $N\%$ of the L1 CPU cache. When the JIT compiles new, newarray or anewarray bytecodes, instead of just incrementing the tlhAlloc pointer, we check whether that pointer is in the reserved N% of the cache. If it is, then the allocation continues as before. If it is not, then the heapAlloc pointer is bumped (incremented) so that it maps to the beginning of L1 cache, and the space in between is marked as used (so that other objects are not allocated in that area). We call this *weak* allocation because although we guarantee that tlhAlloc will be in the reserved section, this is just the beginning of the object, and its end might extend

into the non-reserved section. The TLH space that was not used has to be marked as used because otherwise, GC would get confused about what is in that part of the heap, as it walks the heap (and walking the heap is required for compacting GC schemes).

For example, we can select N to be 25, so that we try to allocate all objects from the first 25% of the cache. Let us assume that an AMD Opteron (Shanghai) processor is used, which has 64KB of L1 data cache, and is 2-way set associative with a 64-byte line. Hence, this cache has 1024 lines. The fact that it is 2-way set associative means that there are effectively only 512 uniquely addressable sets, with 2 lines in each set. In relation to CLR definitions outlined in Section 3.1, this means that we can think of the TLH as being made up of chunks of `chunkSize = 64KB/2 = 32KB`, and with `sectionProportion[0] = sectionProportion[1] = 0.25`. The location where a physical address maps to the cache is determined by its address' least significant bits. In this case, when CLR is enabled, tlhAlloc is forced to have bits 13 and 14 set to 1. Bits 13 and 14 would determine which quarter of the cache the address will be mapped to, because $2\hat{1}5 = 32$KB. This is done in the compiled assembly code using bitwise arithmetic. N, as well as the cache size and associativity are constants that can be changed in the prototype. Figure 4.3 illustrates how objects are allocated in the weak CLR prototype.

One might ask why we would allocate all objects in this way. The idea is for this to be used with generational GC (gencon) that is unaware of CLR (see Section 1.3.2). On local GC, gencon copies all the live objects from one half of the nursery to the other. As a result, the heap is compacted, but it also means that all objects, if they survive a GC, will be spread evenly across the L1 cache. This makes sense in the context of CLR because if they survive a GC, then they are longer-lived objects anyway, and their place is not in the reserved area of the cache. In other words, we are trying to approximate the allocation of short lived objects in reserved areas, which target CLR benefits as described in Sections 3.2.1 and 3.2.2.

The advantage of this prototype is its simplicity: it does not require any additional pointers in the TLH, and there is no modification outside the JIT: If an object needs to get allocated from the VM, it can still be allocated as before, by incrementing the `tlhAlloc` pointer. The disadvantage of this scheme is that objects might occupy non-reserved sections, so cache pollution is significant. In addition, having more than one reserved section is not possible. Also, it offers no control over what objects to reserve, and the heap will fill up very quickly (as outlined in 3.3).

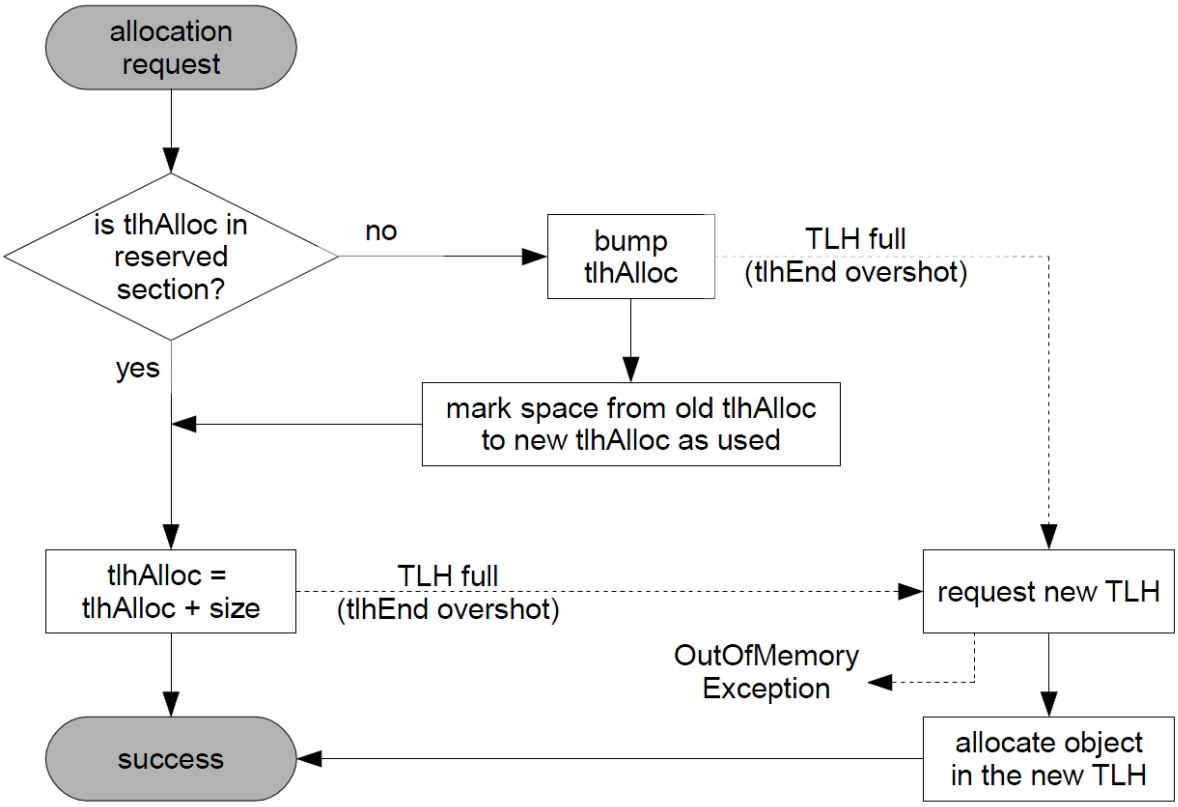

Fig. 4.3 Allocation in the weak CLR prototype

4.3.2 Prototype 2 - "Strong" CLR Reservation

A second prototype was developed that guarantees that a reserved object will be in its entirety in the reserved region. We call this *strong* reservation. It comes at the expense of more overhead for allocation in compiled assembly code, but reserved objects do not pollute the non-reserved sections. This also means that we can safely have more than one reserved section.

In addition to `tlhAlloc`, we add the `tlhReserved` pointer (in terms of the CLR definitions, this would be `tlhAlloc[1]`). Then, non-reserved allocations can be done by incrementing the `tlhAlloc` pointer and reserved allocations are done by incrementing the `tlhReserved` pointer. `tlhReserved` is kept at null if there were no reserved allocations. `tlhAlloc` is always kept above `tlhReserved`, so that if objects need to be allocated from the VM (not the JIT), the conventional scheme of allocation will work without much modification (the `tlhAlloc` can safely be incremented, as it will not overshoot `tlhReserved`). Again, each time that the TLH gets full or the `tlhReserved` pointer is bumped, to prevent having a hole of unallocated space in the middle of the TLH, the area between tlhReserved and the end of the current reserved section is marked as being used (it will be reclaimed at the next GC). Figure 4.4 shows the steps that occur.

In this prototype, when the JIT is compiling a method that allocates an object, it decides based on the object name (or if it is an array, based on the name of the method that contains it), whether to generate code for reserved or normal allocation. Even though Figure 4.4 might suggest that the overhead for reserved allocation is significant, most of that control-flow is for checking limiting cases. In most reserved allocations, what will end up happening is that `tlhReserved` will simply get incremented (after some untaken conditional branches), just like `tlhAlloc` is incremented in the in the non-reserved, conventional case. The behaviour of the compiler is controlled through environment variables and precompiler constants (what objects to reserve, the size of the reserved section, cache associativity, size and others).

The advantage of this prototype is the fact that it does not significantly affect non-reserved allocations, as `tlhAlloc` is always kept above `tlhReserved`. The JVM can function normal as before, and only *if* reserved allocation is needed is the `tlhReserved` pointer initialized. Of course, it is an approximation to CLR with a lot of cache pollution, but due to its low overhead, it might be the most useful one for a production-level JVM.

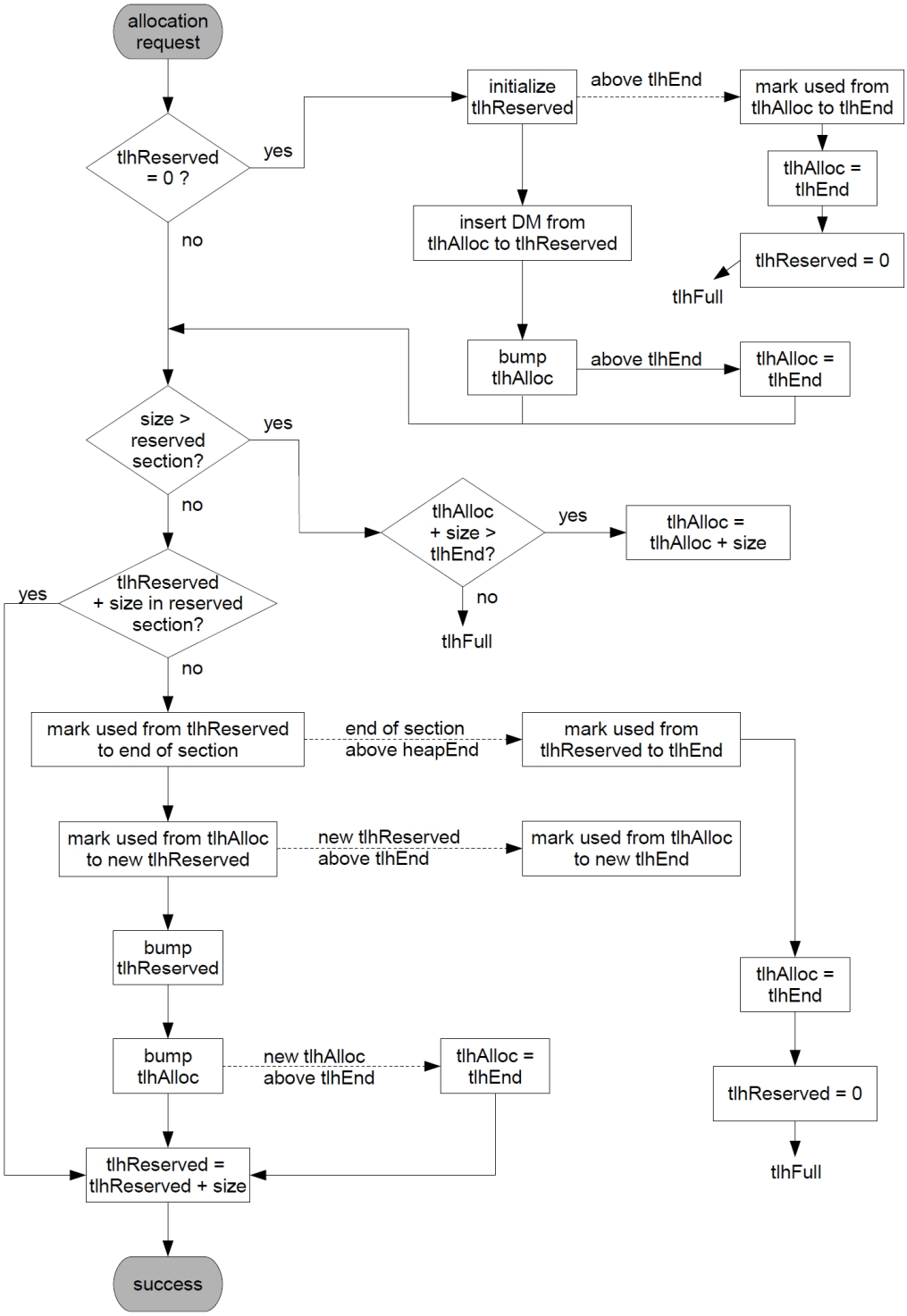

Fig. 4.4 Reserved allocation in the strong CLR prototype

4.3.3 Prototype 3 - "Strong" CLR Reservation and Non-Reservation

The second prototype guaranteed that a reserved object would be in the reserved section, but it did not guarantee that the non-reserved object will not be in the reserved section. In the third prototype, we allocate non-reserved objects only to the non-reserved regions (from the JIT). The objects allocated from the VM are still allocated in the conventional way, and that is possible because `tlhAlloc` is kept above `tlhReserved` at all times. Reserved allocations are done identically as in prototype 2. A flowchart showing how the non-reserved allocation is performed is shown in Figure 4.5. Inspired by the second prototype, it is also possible to allocate *non-reserved* objects in the *weak* way. In that case, we only guarantee that the beginning of the non-reserved object is in the non-reserved section.

The advantage of this prototype is that it minimizes cache pollution (for objects allocated from the JIT). However, each time the TLH gets full or in in interpreted methods, objects are allocated from the VM, and are likely to cause cache pollution.

4.3.4 Other modifications

Apart from the JIT, some modifications were made in the JVM as well. In particular, every time the TLH got full, on clearing the current TLH pointers, `tlhReserved` would be set to 0, and the space between the `tlhReserved` and the beginning of the next non-reserved section would be marked as used space. This was done because the TLH can get full not just from JIT allocations, but also from VM allocations. Also, when there is a global GC, the TLH also gets cleared. Failing to reset the `tlhReserved` pointer would make the JIT use a corrupt pointer next time it does an allocation (from a different TLH). Also, various optional assertions were inserted in the code, such as making sure that `tlhAlloc` is above `tlhReserved` at all times, that marking of the used space always goes from low to high, and that `tlhReserved` does not get modified outside the JIT allocation functions. Counters that record the number of new TLH requests and garbage collections were also implemented.

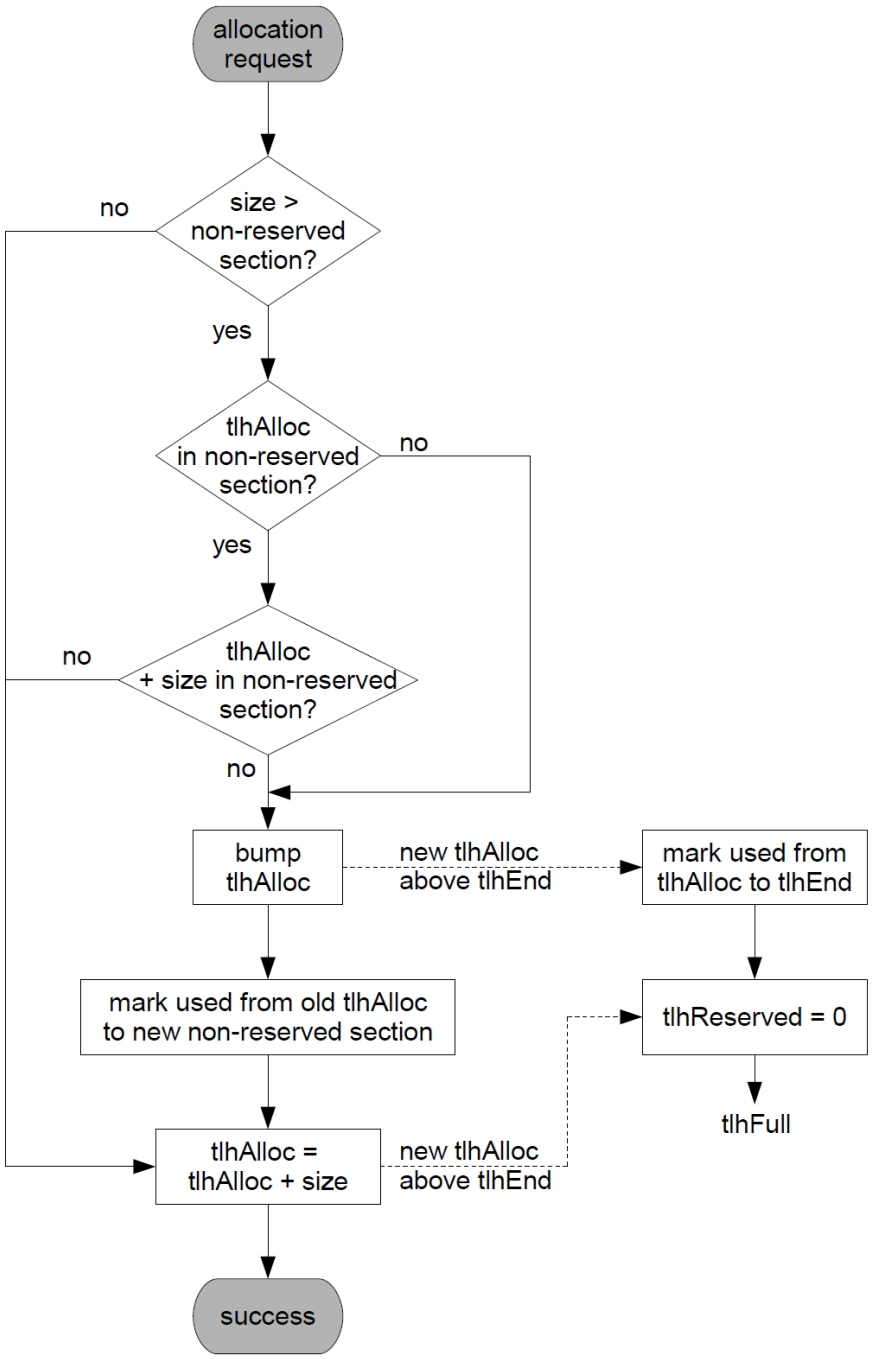

Fig. 4.5 Non-reserved allocation in Prototype 3

Chapter 5

Experimental Results

After the prototypes were developed, we put them to a test in various ways. The first goal was to make sure that CLR can provide a benefit in terms of execution time. For that, we have developed our own benchmarks, in C (unrelated to the JVM prototypes) and in Java (for use with the JVM prototypes). The next goal was to see whether CLR can provide a benefit in standard SPECjvm2008 [76] and SPECjbb2005 [77] benchmarks. Lastly, we measure the overhead that CLR introduces in its current implementation. We begin the chapter by describing the experimental setup used.

5.1 Experimental Setup

Unless otherwise specified, all tests were done on a system with two AMD Opteron (Shanghai) processors, running Windows Server 2003. Its L1 cache size is 64KBytes, and it is 2-way set associative, with a 64-byte line size. Although in total, there are 8 cores (4 in each Opteron), only 4 were used to prevent memory access variations due to NUMA (non-uniform memory access) [50]. In a multiprocessor system, if one processor is waiting for data, it could stall all other processors. NUMA systems try to alleviate this by letting each processor have its own separate memory. The downside of this is that any one processor can have a slower access to other processors' cache lines, because of the overhead of keeping cache coherency. By using only 4 cores, we used only one processor, and hence memory associated with only one socket. AMD's CodeAnalyst software tool was used to profile benchmarks (described in Section 4.2.4). The counters that were interesting to us were: *executed instructions*, number of *memory accesses*, *cache misses* and *lines evicted*. From

these, we may further define the following metrics:

$$\text{miss rate} = \frac{\text{cache misses}}{\text{executed instructions}}$$

$$\text{miss ratio} = \frac{\text{cache misses}}{\text{memory accesses}}$$

$$\text{evicted rate} = \frac{\text{lines evicted}}{\text{executed instructions}}$$

These will be the main metrics in determining whether CLR works or not. Whenever AMD CodeAnalyst was used to profile a run, it was set up to automatically start profiling after a predetermined time, and then to profile for a fixed time. These conditions were always kept the same for the CLR and non-CLR runs, in order to achieve a fair comparison. The tests were also run back-to-back from a batch file. Whenever a claim was made that CLR made an improvement, the test was repeated multiple times to verify the results. The heap memory used was fixed at 1.77GB when using a 32-bit JVM and to 3.54GB when using a 64-bit JVM version. This was done so that the JVM's internal memory management (i.e. adjusting the heap size dynamically based on application memory usage) does not interfere with benchmark scores. Unless otherwise stated, the GC policy used was optthruput.

For all SPECjvm2008 and SPECjbb2005 tests, as we are using a development version of the J9 JVM and Testarossa JIT compiler, IBM Canada has not allowed the publication of absolute scores. Rather, a relative comparison of scores is provided (generally with and without CLR) given in percentages. For both benchmarks, a bigger score is better. This is in contrast to most of the custom benchmarks, where we measure execution time in seconds, and a smaller score is better.

5.2 Proof of Concept

To make sure that the CLR can work at all (under any implementation), we have developed two proof-of-concept C programs that mimic what would happen in CLR prototype 2 when CLR is not enabled, and when CLR is enabled (the main loops of the programs are given

in Appendix A). It was made to determine whether CLR can offer a benefit, and whether CLR can function hardware-wise. The following steps were performed in the program for the non-CLR version:

- Allocate using malloc a continuous area of memory (192 KBytes), and assign it an integer pointer. We can think of it as consisting of 6 chunks.
- Read and write to every 16th integer in the first chunk. This is so that we can bring in one cache line with every access, as a single cache line holds 16 integers (4-bytes each) on the Opteron processor.
- Read and write to every 16th integer in the second chunk.
- Read and write to every 16th integer in the third chunk.
- Repeat this loop of accesses multiple times and measure the time taken.

This can be thought of as creating three linked lists of size equal to one chunk, with elements equal to one cache line size, and then traversing all three linked lists in order. The CLR case is trying to mimic the CLR scheme where 25% of the cache lines are reserved, and where the first array has been allocated to a reserved section (it will span across four reserved sections):

- Allocate 192 KBytes with malloc and assign it an integer pointer.
- Read and write to every 16th integer in the first 25% of the first chunk.
- Read and write to every 16th integer in the first 25% of the second chunk.
- Read and write to every 16th integer in the first 25% of the third chunk.
- Read and write to every 16th integer in the first 25% of the fourth chunk.
- Read and write to every 16th integer in the fifth chunk.
- Read and write to every 16th integer in the sixth chunk.
- Repeat this loop of accesses multiple times and time it.

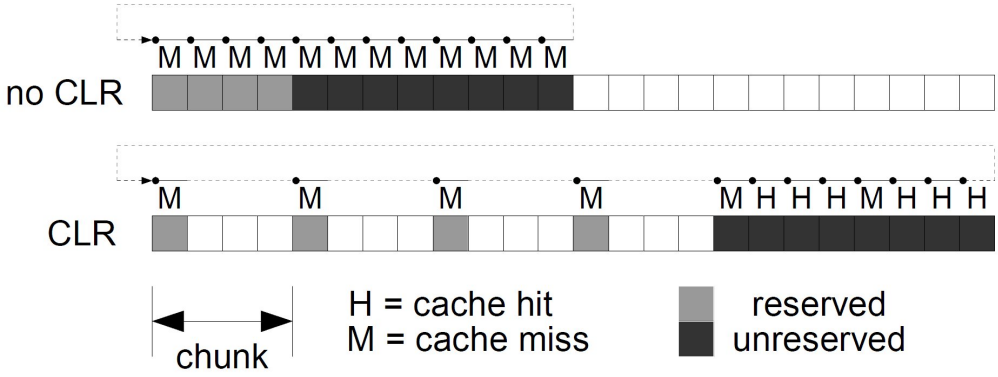

Fig. 5.1 Access pattern in the proof-of-concept C program

If we think about the steady state of these loops, then in the non-CLR case, all of these accesses will be cache misses. There are three chunks being accessed. The first two will completely fill up the cache (because it is 2-way set associative), and when the third one is accessed and brought into the cache, all of these accesses will be misses. The third chunk will evict the first chunk out of the cache, and then when the first chunk is accessed again, we will have misses again. This cycle will repeat to give us only misses. In the case of CLR, in the steady state of the loop, the accesses from the first four chunks will be misses, but 75% of accesses in chunks 5 and 6 will be hits. If we take into account the amount of accessed lines in the chunks (chunks 3, 4, 5 and 6 are only 25% filled with objects), this will mean that half of those accesses will be hits, and the other half misses. Figure 5.1 visually shows this pattern of cache misses and accesses for both cases.

Table 5.1 C program proof-of-concept results (smaller time is better)

option	time (s)	miss rate	miss ratio
no CLR	5.97 (100%)	0.209 (100%)	0.7368 (100%)
CLR (25%)	3.72 (62%)	0.114 (55%)	0.4232 (57%)

Table 5.1 shows the results of the experiments. From the results, we can see that CLR does offer a significant improvement in performance. In theory, the cache misses should have halved, but due to overhead, the miss rate went down only to 55%, which is not far from theory. This translated to a CLR execution time of 62% of the non-CLR execution time. Some of the overhead is due to the evaluation of loop conditions. In either case, this proof-

of-concept program shows that cache misses can translate to a performance improvement, and that CLR can provide this improvement. The C code was compiled using Microsoft's Optimizing Compiler Version 14.00, with an optimization level of 2 (/O2 switch: maximize speed).

5.3 Custom Benchmarks

After the proof-of-concept C program confirmed that CLR can function, a similar microbenchmark was developed in Java. Its job is to evaluate out the Java prototypes and confirm that CLR can provide a benefit in the JVM and the Testarossa JIT. The benchmark performs the following steps:

- A linked list class is created with nodes that were 64 bytes each, the size of one cache line. The nodes are padded with dummy data to increase their size.
- A number of linked lists is created and 512 nodes (one chunk) are added to each of them. Some of them are set up so that nodes are allocated to the reserved areas, some of them not.
- The linked lists are traversed in a specific order (going from the first to the last element). If a reserved linked list is traversed followed by a traversal of a non-reserved linked list, we denote this as "RN".
- Optionally, the data in the nodes (an int) is changed to create a write instruction (in addition to the read).
- This traversal is repeated many times in a timed loop.

This benchmark provides a pattern of accesses very similar to the proof-of-concept C program. Creating a linked list with 512 elements is similar to allocating one chunk of memory with malloc and traversing the linked list is just like reading every 16th integer in the allocated chunk. As before, we reserve 25% of the cache, and prototype 2 is used. Table 5.2 summarizes the results.

From Table 5.2, you can see all the variants of the benchmark that was performed. The results confirmed what was expected. When we had only one linked list, CLR slowed things down significantly (in the R case, the test took twice as long to execute with CLR when

Table 5.2 Custom linked list benchmark performance (smaller time is better)

	read		read/write	
traversal order	no CLR	CLR (25%)	no CLR	CLR (25%)
R	0.91	2.16	0.92	3.17
RR	1.97	4.19	2.05	6.22
N	0.89	0.94	0.92	0.99
NN	2.13	2.14	2.28	2.28
RN	2.25	3.28	2.38	4.75
NR	2.23	3.28	2.36	4.75
NNR	6.45	**4.86**	10.55	**6.48**
NRN	6.48	**4.83**	10.66	**6.81**
RNN	6.45	**4.88**	10.56	**6.81**
RRN	6.47	**5.34**	10.55	**7.83**
RRNN	8.42	**7.00**	14.36	**9.95**
RNRN	8.53	**6.91**	14.50	**9.88**

Values shown represent execution time in seconds. Gencon GC policy, constant total heap size 1.77GB (new space of 1.1GB).

only reading, and 3 times longer when both reading and writing). We would expect that the order of traversal does not matter in steady state of the loops. This is observed from the results, as RN and NR, NRN and RRN, as well as RRNN and RNRN cases have very close scores. Next, it is confirmed that the cache associativity is 2-way because there is a time improvement only once we allocate 3 or more linked lists (the last two fill up the cache entirely, and the third one evicts the first one). The NNR read/write benchmark mirrors the proof-of-concept C program, with the results also being very similar (10.55 seconds without CLR, and 6.48 seconds with CLR, which is 61% of the non-CLR case, compared to 62% in the proof-of-cocept C program). Profiling of the NNR case (Table 5.3) shows that the performance improvement does indeed come from a reduction in cache misses.

Table 5.3 Cache profile of the RNN read/write run on the custom linked list benchmark (score is shown in Table 5.2, more executed instructions is better)

	executed instructions	% of total samples	miss rate	miss ratio
no CLR	68943 (100%)	76.4%	0.136 (100%)	0.152 (100%)
CLR (25%)	149884 (217%)	79.9%	0.066 (48%)	0.097 (64%)

From Table 5.3, it can be seen that the application thread was able to execute 117% more instructions during the measurement window than without CLR. The column showing the percentage of samples is a reminder that the application thread in the JVM is not the only thread that is running: there is also the JVM thread, the GC thread, the JVM profiling thread and the CodeAnalyst thread to collect samples. This proportion improved for the CLR case as well. The fact that the miss ratio went down only to 64% (as opposed to 58% as in the proof-of-concept C progrem as shown in Table 5.1) means that there is more overhead in terms of memory accesses in the JVM implementation of the benchmark than with a bare C program, which is expected.

The custom benchmark demonstrates that CLR can provide a measurable benefit in terms of execution time using the Java prototype that we have created. The next step is to try to exploit opportunities similar to this benchmark in official benchmarks, such as SPECjvm2008 and SPECjbb2005.

5.4 SPECjvm2008

SPECjvm2008 is a suite of benchmarks that was designed to measure performance of the Java Runtime Environment (JRE). From our perspective, it is interesting because it has a wide range of different benchmarks. We have used prototype 1 so that all objects are allocated to reserved areas. The hope is to identify any improvement that CLR can bring, as we do not know which specific objects to reserve. In addition, we have varied the sizes of the reserved sections to see how this will affect performance. Table 5.4 shows the results of one execution.

Table 5.4 Scores on SPECjvm2008 using prototype 1 (bigger score is better)

	CLR (12.5%)	CLR (25%)	CLR (50%)	CLR (75%)	no CLR
startup.helloworld	100	96	102	103	100
startup.compiler.compiler	94	98	101	101	100
startup.compiler.sunflow	100	100	100	102	100
startup.compress	120	97	114	115	100
startup.crypto.aes	109	101	104	101	100
startup.crypto.rsa	105	92	103	109	100
startup.crypto.signverify	109	58	88	71	100
startup.mpegaudio	98	92	105	102	100
startup.scimark.fft	102	99	98	97	100
startup.scimark.lu	100	97	97	109	100
startup.scimark.monte_carlo	100	99	114	98	100
startup.scimark.sor	91	88	93	91	100
startup.scimark.sparse	103	131	99	162	100
startup.serial	96	100	96	98	100
startup.sunflow	66	71	103	82	100
startup.xml.transform	94	97	94	97	100
startup.xml.validation	96	97	94	97	100
compiler.compiler	39	53	79	91	100
compiler.sunflow	31	50	78	90	100
compress	100	96	99	100	100

crypto.aes	106	101	99	112	100
crypto.rsa	97	101	98	98	100
crypto.signverify	98	98	97	99	100
derby	58	80	88	93	100
mpegaudio	101	102	100	102	100
scimark.fft.large	75	81	88	110	100
scimark.lu.large	93	96	96	98	100
scimark.sor.large	101	101	89	137	100
scimark.sparse.large	105	111	101	107	100
scimark.fft.small	100	106	107	107	100
scimark.lu.small	99	99	99	99	100
scimark.sor.small	101	100	101	100	100
scimark.sparse.small	100	100	79	100	100
scimark.monte_carlo	98	99	106	104	100
sunflow	78	96	91	101	100
xml.transform	74	89	94	98	100
xml.validation	70	81	91	95	100
overall	**79.4**	**88.7**	**93.8**	**99.7**	**100**

Scores shown are normalized to the "no CLR" case (where "no CLR" is 100).

From Table 5.4, it is shown that the smaller the reserved section, the worse the CLR performance. More importantly, there is no clear result where CLR provided a benefit. Upon repeating them multiple times, much variation of scores was seen in startup tests because they generally execute for a shorter period of time. It is assumed that random variations in startup conditions (such as what pages are currently in memory) have a larger influence.

When it comes to CLR, the reason why smaller reserved areas produce worse scores is because in prototype 1 we are allocating *all* objects to the reserved areas, so the TLHs will fill up more quickly the smaller the reserved section. For example, with a reserved section of 12.5% , the TLH will fill up about 8 times as quickly (compared to the no CLR case). This will result in more TLH requests and GC time.

Next, specific objects are reserved, using prototype 2, in an attempt to perform more targeted reservations. Specifically, the following objects are reserved: Strings, StringBuffers, StringBuilders and BigDecimals. The char[] array in String, StringBuffer and StringBuilder is also allocated to a reserved section (instead of allocating just the container objects). The reasoning behind reserving these objects is that we have observed empirically that a lot of them are being allocated frequently, so there is a good chance that they are short lived. We are also reserving objects that have been identified as candidates for stack allocation through escape analysis. Escape analysis (EA) is a compiler optimization that can detect variables and objects whose scope is only limited to a certain method body. In those cases, instead of allocating these objects on the heap, they can be allocated on the stack instead. These objects die with the method and allocating them on the stack saves GC time. We disable these stack allocations, but use the results of escape analysis to identify these objects, and reserve them using CLR. They are a typical example of short-lived objects.

Table 5.6 Scores on SPECjvm2008 when reserving specific objects (bigger is better)

options:	run 1	run 2	run 3	run 4
disable EA	no	yes	yes	yes
reserve EA objects	no	yes	yes	yes
reserve Strings	no	yes	yes	yes
reserve StringBuilders	no	yes	yes	yes
reserve StringBuffers	no	yes	yes	yes
reserve BigDecimals	no	yes	yes	yes
overall score	**100**	**98**	**98**	**98**

Scores shown are normalized to the "no CLR" case (where "no CLR" is 100).

Table 5.6 shows the SPECjvm2008 results when reserving specific objects. The overall scores with and without CLR were very similar (the drop in the scores when stack allocations were disabled was expected, and is not related to CLR). However, once again, upon repetition of the tests, CLR was found to make no difference in performance, and any variations of the scores were due to variations in the running conditions of the JIT and the JVM. The full scores were omitted, and only the overall scores are shown. This means that if we are to observe an improvement due to CLR in SPECjvm2008 (and in general),

we have to find a specific opportunity that has to be exploited using CLR. SPECjbb2005 is examined next.

5.5 SPECjbb2005

SPECjbb2005 is a benchmark for evaluating server side Java. It emulates a three-tier client/server system with the emphasis on the middle tier. It uses a number of fictional warehouses to create realistic workloads. Each warehouse runs in a separate thread, and as the number of warehouses is increased, additional CPUs are put to use. Therefore, it also measures scalability of the computer system, apart from its performance. On the system used for benchmarks, because there are 4 cores, the peak is reached with 4 warehouses.

Similar tests were performed as presented for SPECjvm2008 (the results are omitted). Prototype 1 was used to allocate every object to the reserved section and prototypes 2 and 3 were used to reserve String-based, BigDecimal and local objects identified by EA. CLR performance improvement was not seen.

It is clear that a more intelligent way of selecting objects to be reserved is needed if CLR is to yield an improvement in SPECjbb2005 or SPECjvm2008. In an attempt to find CLR opportunities, we have identified a method, *populateXML*, that gets called often during SPECjbb2005, and that might hold an opportunity for CLR.

5.5.1 populateXML

Although we are unable to profile the code for populateXML, as the code for SPECjbb2005 is closed-source, we can describe what it does. Roughly, it performs the following steps:

- It goes through each row in a 2D `char` array (usually of size `char[24][80]`)
- For each of the 24 rows in the array, an instance of a new `String` object is created, with the constructor argument being the current row in the 2D array. Therefore, each `String` will be of length 80.
- Inside the `String` constructor, because Strings are immutable in Java, a new `char` array is created and contents of the row of the 2D array are *copied* to the new array.
- The newly created `String` is used to call another method that does not do anything time-intensive.

Every time populateXML allocates a String, a new array has to be allocated and brought into the cache. These Strings are not zero-lived, but they do consistently die as they are used to fill up a buffer which has a limited capacity. In order to measure the cost of different overheads, we have modified the populateXML method 5 times to create 5 different benchmarks:

- **Benchmark 1:**
  Instead of instantiating the String, a constant String literal is used, of length 80 (an example of a String literal would be: `"abcd"`).
- **Benchmark 2:**
  A constant String literal of length 80 is used in the constructor for instantiating the String (e.g. `new String("abcd")`).
- **Benchmark 3:**
  In addition to what is done in benchmark 2, the contents of the corresponding row in the 2D array are read (and stored into a "dummy" static `char` field). This is to simulate the reading of the row in the copying of the array in the `String` constructor.
- **Benchmark 4:**
  A "dummy" static `char` array of size 80 is created, and used to call the `String` constructor. This way, the benchmark is almost identical to the real SPECjbb2005, but the effect of locality of the 2D array is removed from the score (as only one row is used, 24 times).
- **Benchmark 5:**
  Unmodified SPECjbb2005 benchmark.

The diagnostic measurements (Table 5.7) indicate that a total of 8% of time is spent for the array copy inside the String constructor. This is a significant amount, and if the locality of these array copies is improved, one should see a benefit in performance. The String allocations should be perfect candidates for CLR.

Table 5.8 shows what happened when String objects from the method populateXML and their respective char arrays were allocated to the reserved section (using prototype 2). Compared to the "no CLR" run, the score with CLR was worse, but the cache performance

Table 5.7 Diagnostic benchmarks for calculating overhead in populateXML (bigger score is better)

benchmark	score	source of overhead	contribution
1	112%		
2	110%	String allocation (without the array copy)	2%
3	108%	writing the 2D array for array copy in the String constructor	7%
4	101%	reading the 2D array for array copy in the String constructor	2%
5	100%	locality of the 2D array	1%

Measured with 1 warehouse for 4 minutes, average of 3 runs.
Scores were scaled to the score benchmark 5.

Table 5.8 Cache profiles of SPECjbb2005 when reserving Strings in populateXML (bigger score is better)

	score	miss rate	miss ratio	evicted rate
no CLR	100%	0.0101	0.0238	0.0218
CLR (25%)	85%	0.0098	0.0242	0.0214

Measured with 4 warehouses for 4 minutes, average of 3 runs.
Scores were scaled to the “no CLR” run.

did not improve much. Although the miss rate and the evicted rate have decreased, it was only slight, and the miss ratio (cache misses divided by the number of memory accesses) has gotten slightly worse. The reason for a similar cache profile is likely the following. One char in Java occupies 2 bytes, so an 80 character array occupies 160 bytes. Every time populateXML is called, 24 of those are allocated, for a total of 3840 bytes. If each cache line is 64 bytes, that mens that every time that populateXML is called, 60 lines worth of reserved objects are allocated (and the whole cache contains 1024 lines, with 2-way associativity). These objects will therefore evict 60 cache lines. When CLR is disabled, these 60 lines will be random. With CLR is enabled, these 60 lines will be in the first 25% of the L1 cache. But they will still produce cache misses, no matter where they are. Before populateXML gets called again, the previous reserved array has already been evicted, and then we have another set of misses, no matter where they are. For CLR to work in this case, it needs to evict its *own* reserved objects from the cache. This would be true if populateXML would be called more often. Or, if the array was bigger than 60 lines.

Table 5.9 Cache profiles of SPECjbb2005 when reserving larger Strings in populateXML (bigger score is better)

	score	miss rate	miss ratio	evicted rate
no CLR	100%	0.0093	0.0224	0.0351
CLR (25%)	88%	0.0058	0.0157	0.0225

Measured with 4 warehouses for 4 minutes, average of 3 runs.
Scores shown are normalized to the "no CLR" case (where "no CLR" is 100%).

When prototype 2 was tested with a bigger array, it was clear that the cache misses have decreased, as Table 5.9 shows. Instead of being 24 by 80, the size of the 2D character array was changed to be 120 by 320. However, there was still no improvement in the overall score. The reason for this is the large GC overhead that is incurred.

To try to investigate and minimize the GC overhead, different GC policies were examined (generational and conventional), as well as different sizes of the heap. Table 5.10 shows the results. Two conclusions can be reached. First, increasing the heap size from 1.77GB to 3.54GB has the effect of decreasing the GC overhead introduced by CLR, so that the application thread has more time to run. The reason for this is that a larger heap will run out of memory less frequently than a smaller heap, assuming a constant rate of allocations. Second, generational garbage collection suffers from less overhead than optthruput garbage

Table 5.10 Investigating the effect of different GC policies and heap sizes in SPECjbb2005 (bigger score is better)

		Time spent in: GC, application			
heap size		optthruput		gencon	
1.77 GB	no CLR	14%	83%	5.4%	91%
	no CLR	36%	60%	17%	80%
3.54 GB	no CLR	8.0%	89%	3.3%	93%
	CLR	24%	72%	10%	87%
1.77 GB	no CLR	14%	83%	5.4%	91%
	no CLR	36%	60%	17%	80%
3.54 GB	no CLR	8.0%	89%	3.3%	93%
	CLR	24%	72%	10%	87%

Performance gap (CLR score vs. no-CLR score)		
1.77 GB	68%	89%
3.54 GB	76%	94%

Measured with 4 warehouses for 4 minutes, average of 3 runs.
Scores shown are normalized to the "no CLR" case (where "no CLR" is 100%).

collection. The reason for this is that optthruput does not compact the heap (it is a simple GC policy where the dead objects are simply reclaimed into the free space pool). As a result, the heap gets fragmented heavily, and new free space requests become more difficult. However, even with a large heap and generational GC, the reduction in cache misses due to CLR was not enough to yield a performance benefit. The best CLR score was 94% of the non-CLR score.

To eliminate the overhead of GC, we shift our focus from reserving objects that are frequently allocated, to objects that are frequently accessed. If objects are found to be allocated only once, and written to or read from many times, then they will not cause the heap to overfill if they are allocated to reserved sections. But, each time they are accessed, they will have to be loaded in the cache and evict other objects. If these reserved, frequently accessed objects are large enough, they should evict themselves from the cache, therefore not evicting as many other objects.

At the time of writing, we do not know of a suitable frequently accessed object in SPECjvm2008 or SPECjbb2005. But, we have tested the potential of this idea by introducing a linked list to the populateXML method in SPECjbb2005. Instead of allocating String objects in populateXML in the normal benchmark, we make it traverse a linked list. The same linked list was used as from the previous custom benchmark, only with 256 nodes this time (25% of the L1 cache to match the size of the reserved section). So, instead of allocating 24 Strings and their char arrays, populateXML now traverses this linked list 24 times.

Table 5.11 Cache profiles and scores of SPECjbb2005 where populateXML traverses a single linked list instead of allocating Strings (bigger score is better)

	score	miss rate	miss ratio	evicted rate
no CLR	100%	0.0369	0.0224	0.278
CLR (25%)	130%	0.0196	0.0405	0.244

Measured with 4 warehouses for 4 minutes, average of 3 runs.
Scores shown are normalized to the "no CLR" case (where "no CLR" is 100%).

Table 5.11 shows that CLR does indeed yield a significant benefit (around 30%) in this specialized benchmark. Although SPECjbb2005 was modified, this experiment shows that CLR can yield a benefit if used to reserve certain objects that are frequently accessed, but allocated only once. The challenge is to identify these objects automatically. This

improvement is only seen with the optthruput GC policy. With generational GC, the improvement is not seen because objects are moved during a collection, so any reserved objects get scattered randomly across the cache. Having a CLR-aware GC policy would solve this problem.

5.6 Measuring Overhead

In this section, the overhead that CLR induces in the current implementation is quantified. For this, our tool will be the SPECjbb2005 benchmark with while reserving larger Strings in populateXML (the same as in Table 5.9). We have already investigated this benchmark thoroughly, and it has a simple structure: it tries to do as many operations as possible for a fixed period of time. We can create a simplistic model of the components included in the SPECjbb2005 score:

$$S = S_{base} - O_{alloc} - O_{comp} - O_{TLH} - O_{GC}$$

where S is the SPECjbb2005 score, S_{base} is what the score would be without the allocation, compilation, TLH request and GC overhead, O_{alloc} is the score reduction due to the allocation overhead, O_{comp} is the score reduction due to the compilation overhead (of the allocation function), O_{TLH} is the score reduction due to the TLH requests overhead and O_{GC} is the score reduction due to the GC overhead. If we measure by how much CLR increases each of the overheads, then we will know by how much the base score needs to be improved in order for CLR to give a benefit.

5.6.1 Allocation Overhead

To measure the allocation overhead, duplicate allocation functions have been created that gets compiled (and executed) just before the real allocation functions. The "dummy" allocation functions are modified from the real ones in the following ways:

- None of the TLH pointers are modified. They are loaded into registers and manipulated as usual while in registers, but all stores to them have been changed to stores to some "dummy" pointers.
- Whenever there is a TLH allocation failure (when the TLH gets full), instead of

jumping out of the JIT into the VM to get a new TLH request and proceed with the allocation, we just jump out of the function and proceed with the jitted code.

- Instead of marking parts of the heap as used space, the corresponding instructions are kept, but the stores use "dummy" locations in memory as operands.

The dummy functions should roughly take an equal amount of time (and execute an equal number of instructions) as the real allocation functions. If the real allocation functions are compiled without CLR as usual, the score is given by:

$$S_{noCLR} = S_{base,noCLR} - O_{alloc,noCLR} - O_{comp,noCLR} - O_{TLH,noCLR} - O_{GC,noCLR}$$

But, if the compiled dummy functions are inserted just before the real functions, we will effectively double the allocation overhead, without affecting any other overheads:

$$S_{noCLR,dummy} = S_{base,noCLR} - 2 \times O_{alloc,noCLR} - 2 \times O_{comp,noCLR} - O_{TLH,noCLR} - O_{GC,noCLR}$$

If we are *not* measuring the score during the warmup stage of SPECjbb2005, there is no compilation time, as all the functions that are needed have already been compiled, $O_{comp,noCLR} = 0$. The allocation overhead can now be calculated:

$$S_{noCLR} - S_{noCLR,dummy} = O_{alloc,noCLR}$$

To keep things in proportion, we can scale this overhead as a proportion of the overall score, S_{noCLR}. A similar measurement can be done when CLR is turned on, to get $O_{alloc,noCLR}$, as a proportion of S_{CLR}. The difference between these two percentages will be the extra allocation overhead that CLR introduces.

After performing experiments, Table 5.12 shows the results. We noticed that there was quite a lot of variation in the scores (as reflected by the large standard deviation), but the mean $O_{alloc,noCLR}$ and $O_{alloc,CLR}$ were as expected: $O_{alloc,noCLR}$ was the smallest, followed by $O_{alloc,CLR}$ for prototype 2, $O_{alloc,CLR}$ for prototype 3 using weak reservation, followed by $O_{alloc,CLR}$ by prototype 3 using strong reservation. Hence, we are able to estimate the CLR overheads as 0.6%, 1.3% and 2.1% in the same order.

Table 5.12 Allocation overhead when reserving Strings in populateXML

	normal	dummy	overhead	standard deviation*	extra CLR overhead
no CLR	100%	97.1%	2.9%	8.1%	**0%**
prototype 2	100%	96.5%	3.5%	5.8%	**0.6%**
prototype 3 (weak)	100%	95.7%	4.3%	4.9%	**1.3%**
prototype 3 (strong)	100%	94.9%	5.1%	5.2%	**2.1%**

Scores shown are scaled to the "normal" case (for each prototype)
* of the overhead; 1 warehouse for 4 minutes, average of 13 runs

5.6.2 Compilation Overhead

To measure the compilation overhead, the allocation functions can be compiled twice. This time, the first function is also a "dummy" function, but it does not get executed. Rather, the code generator generates an unconditional jump just before the dummy function's code that skips over the entire function (but, it *will* get compiled). Now the compilation time is doubled, but all other overheads are kept constant. Similarly as before, the compilation overhead can be calculated:

$$
\begin{aligned}
S_{noCLR} &= S_{base,noCLR} - O_{alloc,noCLR} - O_{comp,noCLR} - O_{TLH,noCLR} - O_{GC,noCLR} \\
S_{noCLR,dummy} &= S_{base,noCLR} - O_{alloc,noCLR} - 2 \times O_{comp,noCLR} - O_{TLH,noCLR} - O_{GC,noCLR} \\
& \\
S_{noCLR} - S_{noCLR,dummy} &= O_{comp,noCLR}
\end{aligned}
$$

This time, the measurements have to be taken during the warmup run of SPECjbb2005, to include the compilation time.

From the results given in Table 5.13, we can see that CLR seems to introduce a significant amount of overhead. Compiling the allocation functions twice during the warmup period slowed down the benchmark by only 0.1%. The worst case scenario for CLR is observed in prototype 2 with $O_{comp,CLR}$ being 2.1%. However, it is unexpected that prototype 2 would have more compilation overhead than prototype 3, because prototype 2 uses the non-CLR allocation function for non-reserved allocations. The test scores have a large standard deviation, because they were measured during the warmup stage. During the warmup stage, the order of compilation can significantly affect the score, and this order is

Table 5.13 Compilation overhead when reserving Strings in populateXML

	normal	dummy	overhead	standard deviation*	extra CLR overhead
no CLR	100%	99.9%	0.1%	7.2%	**0%**
prototype 2	100%	97.8%	2.2%	4.4%	**2.1%**
prototype 3 (weak)	100%	99.3%	0.7%	5.7%	**0.6%**
prototype 3 (strong)	100%	98.3%	1.7%	3.5%	**1.6%**

Scores shown are scaled to the "normal" case (for each prototype)
* of the overhead; 1 warehouse for 4 minutes, average of 13 runs

not determenistic. On the other hand, the test has been repeated 13 times. It is possible that the compilation footprint that prototype 2 produces happens to negatively impact the heuristics for compilation of other methods in the warmup run.

5.6.3 New TLH Request Overhead

For measuring the TLH overhead, we can think of it as consisting of two parts: the time needed to satisfy each TLH request, and the number of requests in total:

$$\begin{aligned} O_{TLH,noCLR} &= O_{TLH,noCLR,perrequest} \times N_{TLHrequests,noCLR} \\ O_{TLH,CLR} &= O_{TLH,CLR,perrequest} \times N_{TLHrequests,CLR} \end{aligned}$$

CLR introduces both more TLH requests due to the TLH filling up more quickly, as well as more heap fragmentation. More TLH requests will be reflected by the increased number of TLH requests, and the fact that heap gets more fragmented will be reflected in a higher average time that is needed to fulfil the TLH request. Currently, there is no direct way to measure the amount of time that the GC spends on satisfying TLH requests, so we have added a counter and a timer in the function that refreshes the TLH. Empirically, there were two clear clusters in the time taken for the function to complete, different by several orders of magnitude. The longer times happen when a TLH request triggers a GC collection, while the shorter times indicate that a TLH was successfully found from the free space list. A counter is incremented every time the second (shorter) cluster time sample is encountered. This enabled us to directly calculate $O_{TLH,noCLR}$ and $O_{TLH,CLR}$. We assume that if time

was spent satisfying TLH requests, then that time directly impacts the score.

Table 5.14 TLH request overhead when reserving Strings in populateXML

	TLH requests (in millions)	time per request (nanoseconds)	TLH request overhead)	extra CLR overhead
no CLR	2.26	350	0.33%	**0%**
prototype 2	8.55	346	1.23%	**0.90%**
prototype 3 (weak)	9.25	340	1.31%	**0.98%**
prototype 3 (strong)	8.9	345	1.28%	**0.95%**

1 warehouse for 4 minutes, an average of 3 runs.

The results in Table 5.14 indicate that in all prototypes, TLH requests increase by an order of approximately 4. $O_{TLH,noCLR,perrequest}$ and $O_{TLH,CLR,perrequest}$ do not change much, which leads us to believe that heap fragmentation due to CLR does not impact performance too much. However, extra TLH requests are significant, and they translate to about 1% of time in the benchmark spent serving extra CLR requests.

5.6.4 Garbage Collection Overhead

The garbage collection overhead can be easily measured using a combination of the *-Xverbosegclog* JVM option and IBM Garbage Collection and Memory Visualizer. A GC log is created of the whole run. Since SPECjbb2005 puts timestamps to standard output when each test begins and ends, it is clear which GC collections are relevant to the particular run of interest (the GC verbose log also has time stamps). We isolate those GC collections that we are included in the run, and load them in the GC and Memory Visualizer, a software tool.

The results in Table 5.15 show that the GC overhead was similar in both prototypes 2 and 3. There were about 3 times more GC requests, which is reflected in more time spent in GC (by about 3.3 times). The mean GC pause time did not change much, again indicating that heap fragmentation is not a significant source of overhead. With such high overhead, GC is the primary source of overhead. The GC policy used for these tests was optthruput.

Table 5.15 GC overhead when reserving Strings in populateXML

	number of collections	mean GC pause (ms)	time spent in GC	extra CLR overhead
no CLR	102	48.57	2.07%	**0%**
prototype 2	320	48	6.4%	**4.33%**
prototype 3 (weak)	327	49.17	6.71%	**4.64%**
prototype 3 (strong)	313	49.23	6.42%	**4.35%**

1 warehouse for 4 minutes, an average of 3 runs.
Opthruput GC policy, 1.77GB constant heap size.

Chapter 6

Discussion

This chapter aims to put all the experimental results together and present the current state of the CLR prototype: where it offers an improvement and where it does not. Preliminary results for the proof-of-concept program are presented on other architectures and the chapter ends with suggestions for future research.

6.1 Where CLR offers a benefit (long-lived objects)

In Section 3.2, objects that could potentially benefit from CLR reservation have been identified. CLR has shown a definite improvement with objects that are accessed in infrequent bursts (as shown in Tables 5.2, 5.3 and 5.11). Furthermore, it was shown that this performance improvement comes from cache misses.

The reason why CLR offers a benefit with objects accessed in infrequent bursts is because the overhead is very low. If we have only one allocation function allocating one or a few objects that are accessed in infrequent bursts, then they will just be allocated once (and not die young). This does not cost much in terms of compilation and allocation overhead. If these objects are then accessed often, we can capitalize on the CLR benefit there, without any other allocations. If the objects are not accessed in infrequent bursts, they could still exploit CLR, if we have many (long-lived) objects that are accessed irregularly, but are used for a very short duration. If we have a lot of these kind of objects competing with other objects for cache lines, then it makes sense to put them in the same section. These objects resemble short-lived objects in usage, and long-lived objects in allocation.

In terms of the potential candidates for CLR that we have identified in Section 3.2, we

can say that CLR in its current implementation can be used with:

- Provided that they are long-lived:
  - Objects that are accessed in infrequent bursts
- Provided that they are long-lived and used for short durations (read to or written to only once):
  - Objects that are unlikely to co-exist
  - Mostly written objects
  - Objects in different threads

In the context of CLR, short-lived objects are those objects that either die right after allocation (zero-lived objects), or die before the next garbage collection without being used after allocation. Java has many of those since it has many immutable classes. Long-lived objects are those objects that survive several garbage collections.

6.2 Where CLR overhead is too high (short-lived objects)

When we have tried to test CLR on general short-lived objects that get allocated frequently, it became apparent that CLR in the current implementation introduces a significant amount of overhead (different sources of overhead were introduced in Section 3.3). For *String* objects in SPECjbb2005 that are short lived and get allocated frequently, we incur a performance cost of 0.6-2.1% due to the allocation overhead, 0.6-2.1% due to the compilation overhead, 0.9-1.0% due to the extra TLH request overhead, and 4.3-4.6% due to the GC (optthruput) overhead when looking at these overheads independently. Together (and also including any other overhead that we did not account for), the total overhead cost is 21%-23%. Still, cache misses decrease as presented in Table 5.9. However, the performance improvement due to the reduced amount of cache misses is overshaddowed by the extra overhead. Therefore, objects that are of frequently instantiated types or are coming from frequently executed allocation sites cannot see a CLR benefit yet.

In terms of the potential candidates for CLR that we have identified in Section 3.2, we can say that CLR in its current implementation can be used with:

- Objects of frequently instantiated types

- Objects coming from frequently executed allocation sites

However, it is important to note that if the allocation overhead is reduced enough in the future, CLR should offer a benefit even with these objects.

6.3 CLR for Other Architectures

One detail that one might notice is that so far, we have used CLR only with one CPU (AMD Opteron 8384). An interesting follow-up experiment would be to determine how CLR performs on other CPUs. We took the proof-of-concept program (Section 5.2) and modified it for different CPUs, based on their cache size and associativity. For example, newer Intel CPUs have an associativity of 8. This would mean that we would traverse 9 chunks of memory for the non-CLR case (instead of 3 for the 2-way associativity used before) and 12 chunks of memory for the CLR case (instead of 6).

Table 6.1 shows the results of executing the modified proof-of-concept program (based on their cache parameters) on various CPUs. On all processors, the CLR version of the proof-of-concept program shows an improvement. However, the results might be misleading because the tests were tailored to the specific cache configuration. Although Intel processors have shown a bigger improvement with CLR, AMD processors are more suited for CLR. This is because CLR favours *small cache associativity* and a *large cache size*. This is because the smaller the associativity, the easier it becomes to evict lines from a cache due to conflict misses. Large caches are good for CLR because by reserving certain lines, we are leaving a bigger proportion of other lines intact compared with smaller caches. The reason why Pentium 4 had poor CLR results was because it had a small cache and a relatively small associativity. Of course, a contributing factor might also be the larger penalty for L1 cache misses on an older processor like Pentium 4 (another indication of a memory wall and the need for effective cache management). Nonetheless, results in Table 6.1 do show that investigating CLR for other architectures is something that is worth looking into.

CLR only relies on having a cache memory that has a lower access time than main memory. One question that remains open is how CLR would benefit from hardware support or any architectural changes. Scratchpad memory (as described in Section 2.1.5) is a hardware addition that somewhat resembles CLR. In particular, scratchpad memory can be used for allocating short-lived objects. However, it does not share the same virtual address space as cache memory. Having different areas of main memory that map to specific cache

Table 6.1 Proof-of-concept C program results on different CPUs (smaller time is better)

CPU	no CLR (R)	CLR (R)	no CLR (R/W)	CLR (R/W)
AMD Athlon Xp 2200+ 64KB, 2-way	18391 (100%)	10109 (55%)	17828 (100%)	10046 (56%)
AMD Opteron 870 (MP) 64KB, 2-way	8109 (100%)	5421 (67%)	9781 (100%)	5687 (58%)
AMD Opteron 8384 64KB, 2-way	4624 (100%)	3109 (67%)	6234 (100%)	3703 (59%)
Intel Pentium 4 HT 3.2 8KB, 4-way	1703 (100%)	1593 (94%)	6546 (100%)	6421 (98%)
Intel Core 2 Quad Q6600 32KB, 8-way	1206) (100%)	704 (58%)	1010 (100%)	1948 1948 (52%)
Intel Xeon E5560 32KB, 8-way	686 (100%)	405 (52%)	1092 (100%)	530 (50%)

All CPUs have lines of size 64 bytes. Values shown represent execution time in milliseconds (not comparable across different cache parameters).

lines would help CLR. In this thesis, this is done "artificially" by leaving gaps in the heap. Another way would be if we had more than one cache memory each coupled with its own main memory. We would effectively have several reserved sections, and allocating memory to a specific section would simply be a matter of using the heap located on that specific main memory. An alternative would be to have a single main memory, but change the way the physical address is mapped to cache. For example, addresses coming from one half of main memory could be mapped to the corresponding half of cache memory. This would eliminate the main overhead of the current implementation of CLR, which is excessive memory space consumption due to the unused memory gaps in the heap.

6.4 CLR for Other Programming Languages

CLR is not limited to Java. Java was chosen in our implementation because it creates many short-lived objects, but as the results show, CLR can help with long-lived objects as well (even more so). In general, all programming languages that use dynamic memory management could benefit from CLR.

Object oriented languages such as Java or C++ have the most potential in exploiting CLR, because of the large amount of objects that they create (see Section 1.1). This thesis dealt with a dynamic compiler, which has a profiler that can be used to influence CLR decisions on-the-fly. However, there is nothing stopping static compilers to allocate objects using CLR as well, by using data from static analysis. In addition to turning CLR on or off automatically by the compiler, programmers could also explicitly specify what objects to reserve (perhaps through an alternative language keyword to `new`).

Non-object oriented languages could also use CLR, as long as they use dynamic memory management. The most obvious implementation would be to create a special version of the `malloc` function that allocates memory from reserved sections. Then, each time `malloc` is called, a heuristic check could be performed that determines whether to use the CLR version of `malloc` or not. This could also allow the programmer to directly call the special version of `malloc` to enforce CLR.

Using CLR with *functional* languages [78] is also possible. Examples of functional languages include Scheme, Erlang and Haskell. Under the hood, functional languages use dynamic memory management and a heap, but they try to hide this from the user as much as possible. Functional languages allocate memory in terms of function frames, as functions

are called. Recursion is very common in functional languages, so it is easy to imagine a scenario where a large amount of function frames would evict all data from the cache. It this situation, it would probably be beneficial to allocate these function frames using CLR.

6.5 Future Directions

Future work should focus on two areas. First, the allocation overhead should be reduced to an acceptable level in order to use CLR for allocation of many short-lived objects. The main source of overhead is the garbage collection overhead, due to the increased usage of the heap. With the current prototype using the J9 JVM, there is very little we can do from the JIT, without considerable modification of the VM/GC. However, it would definitely be possible. One idea could be to have a separate region of the heap *just* for reserved allocations. When this area gets full, a local GC is performed only on that region. Hopefully, all these objects are short-lived, and most of them would die. If not, they can be moved to the conventional area of the heap. We could think of this as pre-nursery space for the nursery (in generational GC schemes). Prototype 1 had a similar concept, only it used the real nursery and allocated *all* objects to the reserved area. If we had this special region of memory for reserved allocations, the allocation overhead should also increase. This is because the CLR allocation pointer does not have to be related to the conventional TLH pointer, and a lot of checks would be eliminated (including guaranteeing that the CLR pointer is above the TLH allocation pointer and marking the heap area as used when these pointers are bumped).

The second area in which CLR should go from here is to try to automatically identify the objects that we already know can benefit from CLR in the current prototypes. We have shown that CLR can help when traversing a linked list. This could be generalized to other objects that have a similar memory access pattern (and other patterns where CLR helps could be identified). Possible starting points include identifying objects that are the same size as one cache line, objects that contain a method with a reference to another object that is next to it in memory, or identifying a large amount of long-lived objects that get accessed together.

Chapter 7

Conclusion

The main contribution of this thesis is the presentation of Cache Line Reservation (CLR), a novel scheme for allocating objects designed to reduce data cache misses. Several prototypes that use CLR were developed and tested using the JIT compiler in J9 JVM. Using these prototypes, it is determined where CLR offers a benefit. In the cases where CLR does not, it is due to the overhead introduced, and this overhead is measured.

Chapter 1 provides background information, and explains the motivation behind the CLR allocation scheme: the ever-increasing memory wall between different cache hierarchies. This means that the penalty of cache misses is a performance bottleneck, and object oriented languages such as Java only contribute to the problem. Chapter 2 examines some existing software and hardware methods that are used to reduce cache misses.

Chapter 3 provides a description of CLR in detail, presented as a high-level general definition without implementation details. We examine what objects could be candidates for reserved allocation, as well as what overhead is introduced. CLR as a concept is not limited to JVMs or even dynamic compilers.

Chapter 4 describes the details of our CLR implementation for Java, using a JIT compiler in the J9 JVM. We have developed three prototypes, all of which are a variation of the general CLR scheme. We also present the tools used in development. We have developed a number of tests to evaluate CLR with our implementation in Chapter 5. A proof-of-concept C program shows a definite performance opportunity on our architecture when using CLR. Custom Java microbenchmarks are presented that show the benefit of CLR in traversals of linked lists. In SPECjvm2008 and SPECjbb2006, the overhead was

too large for a performance improvement to appear when reserving short-lived objects. We continue by measuring this overhead, and show that the main source of overhead is the increased GC time, due to the heap filling up too quickly when allocating reserved objects.

In Chapter 7, we examine how to minimize CLR overhead. The results seen in custom benchmarks indicate that CLR should concentrate on reserving long-lived objects that are allocated once but accessed in short bursts, as well as long-lived objects that are allocated once and used irregularly for shorts periods of time. This eliminates the allocation and GC overhead, as we only allocate these objects once. Future work should focus on trying to identify these objects automatically, as well as reducing overhead so that CLR can be used for allocating many short-lived objects as well. We present some proof-of-concept results to show that CLR is able to operate on a number of different CPUs.

Traditionally, CPU caches were managed exclusively by hardware. In the future, as the size of caches increase, it will become worthwhile for compilers to start performing advanced optimizations catered to specific cache configurations. Cache line reservation is a simple idea that can offer benefits, but making it operate consistently will be a challenge that the compiler community will hopefully overcome in the future.

Appendix A

Proof Of Concept C Programs

```
int i,j, start, time_taken_millis;
int * dataptr; int dummy = 0;
dataptr = (int*) malloc (6*CHUNK_SIZE);
```

Listing A.1 Proof-of-concept program without CLR (2-way cache R/W)

```
start = clock();
for (j = 0; j<ITERATIONS; j++)
{
  for (i = 0; i<CHUNK_SIZE/INT_SIZE; i=i+LINE_SIZE/INT_SIZE)
  {
     dataptr[i]=dataptr[i]+1;  // all misses
  }
  for (i = CHUNK_SIZE*1/INT_SIZE; i<CHUNK_SIZE*2/INT_SIZE; i=i+LINE_SIZE/
     INT_SIZE)
  {
     dataptr[i]=dataptr[i]+2;  // all misses
  }
  for (i = CHUNK_SIZE*2/INT_SIZE; i<CHUNK_SIZE*3/INT_SIZE; i=i+LINE_SIZE/
     INT_SIZE)
  {
     dataptr[i]=dataptr[i]+3;  // all misses
  }
}
time_taken_millis = (int)((clock()-start)*1E3/CLOCKS_PER_SEC);
printf("without␣CLR:␣%d\n", time_taken_millis);
```

Listing A.2 Proof-of-concept program without CLR (2-way cache R/W)

```
start = clock();
for (j = 0; j<ITERATIONS; j++)
{
  for (i = 0; i<CHUNK_SIZE/4/INT_SIZE; i=i+LINE_SIZE/INT_SIZE)
  {
    dataptr[i]=dataptr[i]+1;  // all misses
  }
  for (i = CHUNK_SIZE/INT_SIZE; i<(CHUNK_SIZE+CHUNK_SIZE/4)/INT_SIZE; i=i
     +LINE_SIZE/INT_SIZE)
  {
    dataptr[i]=dataptr[i]+1;  // all misses
  }
  for (i = CHUNK_SIZE*2/INT_SIZE; i<(CHUNK_SIZE*2+CHUNK_SIZE/4)/INT_SIZE;
      i=i+LINE_SIZE/INT_SIZE)
  {
    dataptr[i]=dataptr[i]+1;  // all misses
  }
  for (i = CHUNK_SIZE*3/INT_SIZE; i<(CHUNK_SIZE*3+CHUNK_SIZE/4)/INT_SIZE;
      i=i+LINE_SIZE/INT_SIZE)
  {
    dataptr[i]=dataptr[i]+1;  // all misses
  }
  for (i = CHUNK_SIZE*4/INT_SIZE; i<CHUNK_SIZE*5/INT_SIZE; i=i+LINE_SIZE/
     INT_SIZE)
  {
    dataptr[i]=dataptr[i]+2;  // 3/4 of these should be hits
  }
  for (i = CHUNK_SIZE*5/INT_SIZE; i<CHUNK_SIZE*6/INT_SIZE; i=i+LINE_SIZE/
     INT_SIZE)
  {
    dataptr[i]=dataptr[i]+3;  // 3/4 of these should be hits
  }
}
time_taken_millis = (int)((clock()-start)*1E3/CLOCKS_PER_SEC);
printf("with␣CLR:␣␣␣␣␣%d\n", time_taken_millis);
```

References

[1] J. Kim and Y. Hsu, "Memory system behavior of java programs: methodology and analysis," in *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. Santa Clara, California, United States: ACM, 2000, pp. 264–274.

[2] H. Inoue, D. Stefanovic, and S. Forrest, "On the prediction of java object lifetimes," *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 880–892, 2006.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, May 2002.

[4] I. Bilicki, V. Sundaresan, D. Maier, N. Grcevski, and Z. Zilic, "Cache line reservation: Exploring a scheme for Cache-Friendly object allocation," in *Proceedings of the 2009 conference of the center for advanced studies on collaborative research: meeting of minds*, 2009.

[5] P. Dubey, *A Platform 2015 Workload Model, Recognition, Mining and Synthesis Moves Computers to the Era of Tera*. Microprocessor Technology Lab, Intel Corporation, 2005, white Paper. [Online]. Available: http://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf

[6] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Toronto, Ontario, Canada: ACM, 1991, pp. 30–44.

[7] O. Temam, C. Fricker, and W. Jalby, "Cache interference phenomena," in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. Nashville, Tennessee, United States: ACM, 1994, pp. 261–271.

[8] J. A. G. Pulido, J. M. S. Prez, and J. A. M. Zamora, "An educational tool for testing hierarchical multilevel caches," *SIGARCH Comput. Archit. News*, vol. 24, no. 4, pp. 11–15, 1996.

[9] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, Seattle, WA, USA, pp. 364–373.

[10] P. Viana, A. Gordon-Ross, E. Keogh, E. Barros, and F. Vahid, "Configurable cache subsetting for fast cache tuning," in *Proceedings of the 43rd annual Design Automation Conference*. San Francisco, CA, USA: ACM, 2006, pp. 695–700.

[11] T. Sheu, Y. Shieh, and W. Lin, "The selection of optimal cache lines for microprocessor-based controllers," in *[1990] Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture*, Orlando, FL, USA, 1990, pp. 183–192.

[12] J. M. Velasco, D. Atienza, and K. Olcoz, "Exploration of memory hierarchy configurations for efficient garbage collection on high-performance embedded systems," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI*. Boston Area, MA, USA: ACM, 2009, pp. 3–8.

[13] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Finding optimal l1 cache configuration for embedded systems," in *Asia and South Pacific Conference on Design Automation, 2006.*, Yokohama, Japan, 2006, pp. 796–801.

[14] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, 1999.

[15] C. Zhang, X. Zhang, and Y. Yan, "Multi-column implementations for cache associativity," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, Austin, TX, USA, pp. 504–509.

[16] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th annual international symposium on Computer architecture*. San Diego, California, United States: ACM, 1993, pp. 169–178.

[17] R. Subramanian, Y. Smaragdakis, and G. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, USA, 2006, pp. 385–396.

[18] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[19] I. K. Chen, C. Lee, and T. Mudge, "Instruction prefetching using branch prediction information," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, Austin, TX, USA, pp. 593–601.

[20] J. Baer and T. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. Albuquerque, New Mexico, United States: ACM, 1991, pp. 176–186.

[21] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems - ASPLOS-V*, Boston, Massachusetts, United States, 1992, pp. 62–73.

[22] H. Peter, "Introduction to the cell broadband engine - white paper," Nov. 2005.

[23] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of Scratch-Pad memory in embedded processor applications," in *Proceedings of the 1997 European conference on Design and Test*. IEEE Computer Society, 1997, p. 7.

[24] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proceedings of the 38th annual Design Automation Conference*. Las Vegas, Nevada, United States: ACM, 2001, pp. 690–695.

[25] C. Lebsack and J. Chang, "Using scratchpad to exploit object locality in java," in *2005 International Conference on Computer Design*, San Jose, CA, USA, 2005, pp. 381–386.

[26] K. F. Chong, C. Y. Ho, and A. S. Fong, "Pretenuring in java by object lifetime and reference density using Scratch-Pad memory," in *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, Napoli, Italy, 2007, pp. 205–212.

[27] S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Use of local memory for efficient java execution," in *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, 2001, pp. 468–473.

[28] "PlayStation - wikipedia, the free encyclopedia," Aug. 2009. [Online]. Available: http://en.wikipedia.org/wiki/PlayStation

[29] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. Las Vegas, Nevada, United States: ACM, 1997, pp. 171–182.

[30] J. Kalamatianos, A. Khalafi, D. Kaeli, and W. Meleis, "Analysis of temporal-based program behavior for improved instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 168–175, Feb. 1999.

[31] J. Kalamatianos and D. Kaeli, "Accurate simulation and evaluation of code reordering," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, 2000, pp. 13–20.

[32] S. Bartolini and C. A. Prete, "Optimizing instruction cache performance of embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 934–965, 2005.

[33] N. Gloy and M. D. Smith, "Procedure placement using temporal-ordering information," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 977–1027, 1999.

[34] D. Genius, "Handling cross interferences by cyclic cache line coloring," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, Paris, France, 1998, pp. 112–117.

[35] A. Ramirez, J. Larriba-Pey, and M. Valero, "The effect of code reordering on branch prediction," in *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, 2000, pp. 189–198.

[36] X. Huang, S. M. Blackburn, D. Grove, and K. S. McKinley, "Fast and efficient partial code reordering: taking advantage of dynamic recompilatior," in *Proceedings of the 5th international symposium on Memory management.* Ottawa, Ontario, Canada: ACM, 2006, pp. 184–192.

[37] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation.* White Plains, New York, United States: ACM, 1990, pp. 16–27.

[38] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 5, pp. 1–46, 2009.

[39] X. Huang, B. T. Lewis, and K. S. McKinley, "Dynamic code management: improving whole program code locality in managed runtimes," in *Proceedings of the 2nd international conference on Virtual execution environments.* Ottawa, Ontario, Canada: ACM, 2006, pp. 133–143.

[40] "Jikes RVM - home," Aug. 2009. [Online]. Available: http://jikesrvm.org/

[41] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems.* Santa Clara, California, United States: ACM, 1991, pp. 40–52.

[42] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Ph.D. dissertation, Rice University, 1989.

[43] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of the 18th annual international symposium on Computer architecture - ISCA '91*, Toronto, Ontario, Canada, 1991, pp. 43–53.

[44] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th annual international symposium on Microarchitecture - MICRO 24*, Albuquerque, New Mexico, Puerto Rico, 1991, pp. 69–73.

[45] C. Luk and T. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, Feb. 1999.

[46] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. Montreal, Canada: ACM, 1984, pp. 233–246.

[47] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. Santa Clara, California, United States: ACM, 1991, pp. 63–74.

[48] A. C. McKellar and E. G. Coffman, "Organizing matrices and matrix operations for paged memory systems," *Communications of the ACM*, vol. 12, no. 3, pp. 153–165, 1969.

[49] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. La Jolla, California, United States: ACM, 1995, pp. 279–290.

[50] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for NUMA memory management," in *Proceedings of the twelfth ACM symposium on Operating systems principles*. ACM, 1989, pp. 19–31.

[51] G. Rivera and C. Tseng, "Data transformations for eliminating conflict misses," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. Montreal, Quebec, Canada: ACM, 1998, pp. 38–49.

[52] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New Orleans, Louisiana, United States: ACM, 1999, pp. 616–622.

[53] M. Liu, Q. Zhuge, Z. Shao, and E. H. Sha, "General loop fusion technique for nested loops considering timing and code size," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems - CASES '04*, Washington DC, USA, 2004, p. 190.

[54] B. Girodias, "Optimisation des memoires dans le flot de conception des systemes multiprocesseurs sur puces pour des applications de type multimedia," Ph.D. dissertation, Universite de Montreal, 2009.

[55] M. Wolf, D. Maydan, and D. Chen, "Combining loop transformations considering caches and scheduling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Paris, France, pp. 274–286.

[56] C. S. Lebsack and J. M. Chang, "System level perspective on object locality," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. San Diego, CA, USA: ACM, 2005, pp. 244–245.

[57] M. Hirzel, J. Henkel, A. Diwan, and M. Hind, "Understanding the connectivity of heap objects," in *Proceedings of the 3rd international symposium on Memory management*. Berlin, Germany: ACM, 2002, pp. 36–49.

[58] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Chicago, IL, USA: ACM, 2005, pp. 129–142.

[59] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. Atlanta, Georgia, United States: ACM, 1999, pp. 1–12.

[60] T. M. Chilimbi and J. R. Larus, "Using generational garbage collection to implement cache-conscious data placement," in *Proceedings of the 1st international symposium on Memory management*. Vancouver, British Columbia, Canada: ACM, 1998, pp. 37–48.

[61] P. R. Wilson, M. S. Lam, and T. G. Moher, "Effective "static-graph" reorganization to improve locality in garbage-collected systems," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Toronto, Ontario, Canada: ACM, 1991, pp. 177–191.

[62] ——, "Caching considerations for generational garbage collection," in *Proceedings of the 1992 ACM conference on LISP and functional programming*. San Francisco, California, United States: ACM, 1992, pp. 32–42.

[63] M. B. Reinhold, "Cache performance of garbage-collected programs," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. Orlando, Florida, United States: ACM, 1994, pp. 206–217.

[64] A. Diwan, D. Tarditi, and E. Moss, "Memory system performance of programs with intensive heap allocation," *ACM Trans. Comput. Syst.*, vol. 13, no. 3, pp. 244–273, 1995.

[65] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh, "Exploiting prolific types for memory management and optimizations," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Portland, Oregon: ACM, 2002, pp. 295–306.

[66] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald, "Thread-local heaps for java," in *Proceedings of the 3rd international symposium on Memory management*. Berlin, Germany: ACM, 2002, pp. 76–87.

[67] J. Bonneau and I. Mironov, "Cache-Collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, 2006, pp. 201–215. [Online]. Available: http://research.microsoft.com/pubs/64024/aes-timing.pdf

[68] M. O'Hanlon and A. Tonge, "Investigation of Cache-Timing attacks on AES," 2005. [Online]. Available: http://www.computing.dcu.ie/research/papers/2005/0105.pdf

[69] C. Percival, "Cache missing for fun and profit," 2005. [Online]. Available: http://www.daemonology.net/papers/htt.pdf

[70] "IBM j9 - wikipedia, the free encyclopedia," Aug. 2009. [Online]. Available: http://en.wikipedia.org/wiki/IBM_J9

[71] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan, "Java just-in-time compiler and virtual machine improvements for server and middleware applications," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*. San Jose, California: USENIX Association, 2004, pp. 12–12.

[72] "Debugging tools for windows - overview," Aug. 2009. [Online]. Available: http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[73] "IBM - using IBM HeapAnalyzer to analyze java heap usage and detect possible java heap leak," Aug. 2009. [Online]. Available: http://www-01.ibm.com/support/docview.wss?rs=180&uid=swg21190608

[74] "developerWorks : IBM monitoring and diagnostic tools for java - garbage collection and memory visualizer version 2.3," Aug. 2009. [Online]. Available: http://www.ibm.com/developerworks/java/jdk/tools/gcmv/

[75] "AMD CodeAnalyst for windows," Aug. 2009. [Online]. Available: http://developer.amd.com/cpu/CodeAnalyst/codeanalystwindows/Pages/default.aspx

[76] "SPECjvm2008," Aug. 2009. [Online]. Available: http://www.spec.org/jvm2008/

[77] "SPEC JBB2005," Aug. 2009. [Online]. Available: http://www.spec.org/jbb2005/

[78] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.