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

Ivan Bilicki Vijay Sundaresan Daryl Maier Nikola Grčevski Željko Žilic
McGill University IBM Canada IBM Canada IBM Canada McGill University
ivan.bilicki@mail.mcgill.ca; {vijaysun | maier | nikolag}@ca.ibm.com; zeljko.zilic@mcgill.ca

Abstract

We present 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 don't evict other objects that will be needed later. We discuss what kinds of allocations could benefit from CLR, as well as sources of overhead. Basic prototypes were implemented in the IBM® J9 Java™ virtual machine (JVM) and its Testarossa just-in-time (JIT) compiler. We present preliminary results, which show that CLR can offer a benefit in specialized microbenchmarks, but not yet in SPECjbb2005 and SPECjvm2008. Furthermore, we discuss the limitations of the current implementation and suggest areas for improvement. CLR is not limited to Java applications, and we hope to see it developed for other compilers.

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 [2], or even instantly [3]; 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 paper is provided in [1].

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 some low order bits of the memory 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 paper, 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 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 paper 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.

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 preliminary results on custom microbenchmarks where CLR proves to be beneficial, and discuss SPECjbb2005 and SPECjvm2008 results, where CLR does not offer an improvement yet. The paper concludes with a discussion of the limitations of the current implementation and ideas for future development.

[1]Patent pending (IBM Canada Ltd)

2 Related Work

Cache misses have been thoroughly researched [1]. There are three types: compulsory misses, capacity misses and conflict misses. Conflict misses can be reduced by changing the line replacement policy, cache associativity, or the way in which data is organized [4]. These are the misses that we are trying to reduce in this paper. There is nothing we can do about compulsory and conflict misses for a fixed cache size. We now explore different software and hardware approaches that can be used to optimize cache performance.

2.1 Hardware Approaches

Cache parameters in terms of total size, line size, and associativity can be changed to see how they affect cache misses [2, 5]. This is typically done by collecting different program traces, and then running them through a cache simulator [6, 7]. Caches in embedded systems can be designed using this technique, where another goal is also low power consumption [8].

More sophisticated schemes of caches have been suggested in hardware, such as a *skewed* cache, which doesn't have a fixed associativity. For example, it could have two lines in each set; the first line is direct mapped, but the second line is mapped using a hash function [9]. There has also been work involving non-trivial line replacement policies [10].

Another related concept is using s*cratchpad memory*. Scratchpad memory [11] is a piece of memory close to the CPU that acts similar to L1 cache memory. However, data in scratchpad memory does not have to be present in main memory. This makes scratchpad memory suitable for allocation of short-lived objects [12]. 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. Objects that are longer-lived but are accessed frequently can also benefit from being allocated to scratchpad memory, a process known as *pretenuring* [13].

2.2 Software Approaches

There has been work done with code/procedure reordering to improve instruction cache performance [14, 15]. 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 coloring algorithms can be used to decide where exactly to place them. This approach can also be used to organize arrays in data caches, as opposed to just instruction caches [16].

Software *prefetching* is a technique where the compiler inserts extra prefetch intstructions in compiled code [17]. A prefetch instruction brings data to the cache similar to a load or a store, but it does not do anything with it. This way, when the data is needed later, it will already be in the cache. Prefetching can also be done automatically in hardware by looking at future memory requests.

In the area of scientific computing, where many computations on large amounts of data are performed, cache performance of specific frequently executed loops is important (e.g. matrix multiplication). Various schemes to restructure the loops to be more cache-friendly have been developed [18, 19]. For example, *tiling* reduces the volume of data worked with in each loop iteration, so that it can fit into the cache.

Work has been done to allocate certain objects to specific portions of main memory. The full heap can be partitioned into multiple heaps, where each heap is used for some selected objects [20, 21]. Alternatively, 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 [22, 23]. These schemes improve object locality in memory, which in turn has the effect of improving cache performance.

Another way to improve object locality and reduce the overhead of object allocation is to use a specific thread local heap (TLH) when allocating objects from a specific thread [24]. This approach primarily aims at eliminating the need for synchronization at every allocation in the presence of multiple threads (as would be the case if there was one heap allocation area for all threads). This can be done because a chunk of memory is assigned for exclusive use by only one thread.

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 locality and reducing cache misses. To our knowledge, there is no prior work that mentions this approach of object allocation.

3 Cache Line Reservation

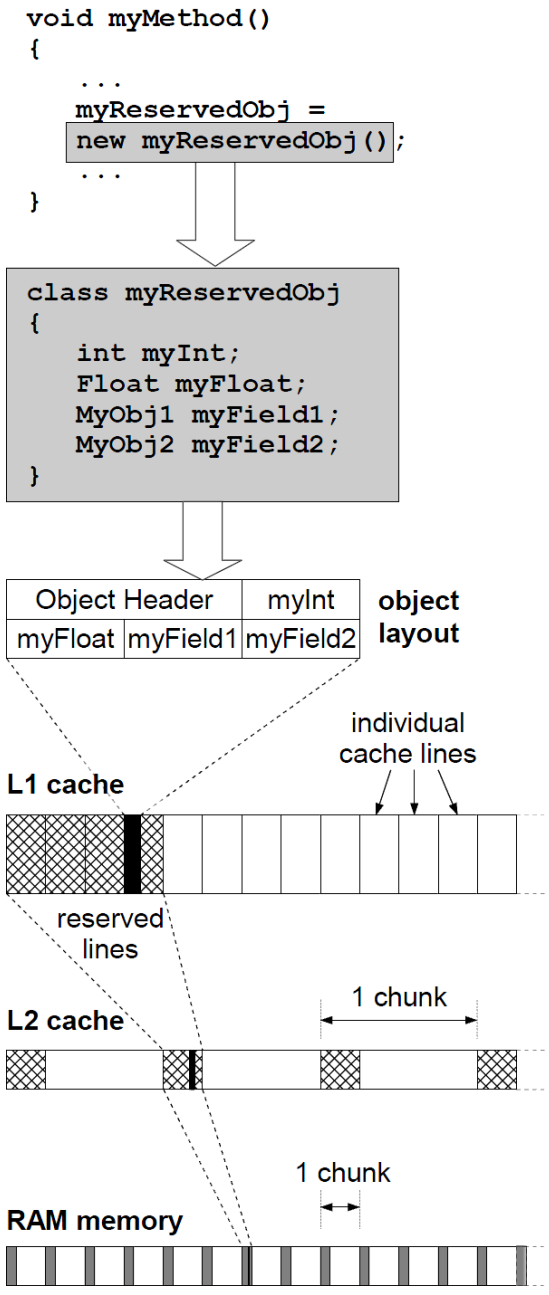

Figure 1: An Overview of CLR

A simple allocation scheme used conventionally by JIT compilers is based on each thread allocating objects using thread local heaps (TLHs). A TLH is a contiguous region of memory of a size controlled by the JIT. The TLH has two pointers that are of interest to us: an allocation pointer, *tlhAlloc*, and a pointer marking the end of the TLH, *tlhTop*. When an object (or an array of primitives) needs 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 about 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 paper, the idea is to divide each TLH into an unreserved section and one (or more) reserved sections. Selected allocations are performed only from designated reserved sections whereas all other allocations are done from the unreserved section. 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. Each TLH should be conceptually viewed as being partitioned into smaller chunks, where each smaller chunk has a size represented by *chunkSize* = (total size of the cache in bytes)/A (where A is the associativity of the cache). 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. Figure 1 visually shows the concept of CLR, when reserving a Java object.

A practical way to implement CLR with N reserved sections would be to define the following variables:

reservedSections
(the number of reserved sections)
sectionProportion[reservedSections]
(the proportion of each section in a chunk)
tlhAlloc[reservedSections+1]
(allocation pointers)
tlhTop[reservedSections+1]
(allocation section tops)
tlhHighestAllocIndex
(index of the highest tlhAlloc entry)
tlhStart
(start of the whole TLH)

tlhEnd
(end of the whole TLH)

When a new TLH is obtained, these variables are initialized, as shown in Figure 2.

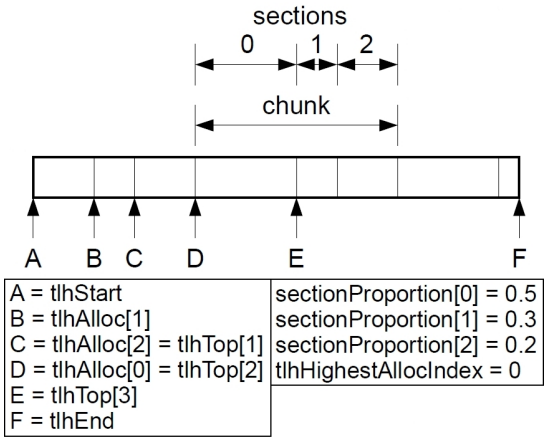

Figure 2: CLR pointer initialization

Allocations can be done by incrementing the corresponding tlhAlloc pointer, depending on what section we want to use. We have to keep track of the tlhTop pointers as well, so that we do not overshoot into another section. In cases where the size of the object is too large for its section, it can be allocated using the highest of the tlhAlloc pointers (which is what tlhHighestAllocIndex is used for). The number of reserved sections and their proportion of the chunks can be a dynamically changing parameter. Other modifications might be needed, such as teaching the GC about CLR when moving objects. In addition, other levels of CLR could be implemented in L2 or L3 caches.

3.1 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, and assign them to a specific reserved section. Here, we explore the possible scenarios where CLR could offer a benefit, and determine how to identify them so that we can (re)compile the allocation code to use CLR.

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 these objects are likely to be short lived. One extreme would be allocating a lot of zero-lifetime objects. 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 (which is 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.

Frequently executed allocation sites
If we have a section of code that allocates certain objects, and this section gets executed often, then these objects allocated from *this* site are likely to be 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.

Objects 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 lifetimes.

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 from or written to in isolation, then they would evict all the lines from the cache when this happens. If they occupy only a specific part of the cache, then they would evict fewer of the other objects from the cache.

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 an 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.

Objects in different threads
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.

3.2 Limitations of CLR

The CLR allocation scheme comes at a certain cost. This cost will depend on how CLR is implemented, as well as how often reserved allocations occur. Although CLR is not necessarily limited to the Java programming language, from the perspective of a JVM that uses THLs, some sources of overhead are as follows.

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.

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 GC.

Reduced non-reserved cache capacity
When reserved sections are allocated, they prevent their cache lines for non-reserved objects. This can actually increase cache misses.

More expensive allocation
Allocation of reserved objects is not as straightforward as incrementing the tlhAlloc pointer in the TLH any more. In addition to more execution time, the size of the compiled methods that use reserved allocation will be bigger.

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.

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 or generational GCs), the GC might have to be modified such that the reserved objects are moved 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 would have to be developed to detect 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 can provide, and under what circumstances. This is the topic of discussion of the rest of this paper.

4 Design & Implementation

The JVM we used is the IBM J9 along with the Testarossa (TR) JIT compiler. A program written in the Java is first compiled to *Java bytecode*. Then the bytecode is traditionally interpreted by the JVM.

Java objects are allocated when the following bytecodes are encountered: *new*, *newarray,* and *anewarray* (allocates an array of references). The allocations can happen through the following paths: from methods that the JIT compiled, from the JVM when the JIT allocation fails (if the TLH gets full or the object is too large for TLH allocation), or from interpreted methods by the JVM. We are interested in the JIT path.

The JIT compiler helps by compiling frequently executed methods to native instructions. Since CLR focuses on frequently allocated and/or frequently used objects, they are likely to be allocated from the compiled code. Therefore, we have modified the JIT compiler so that it can compile methods that allocate objects according to the CLR scheme. All three prototypes that were developed are a simplification of the CLR definition outlines in Section 3.

4.1 "Weak" CLR – Prototype 1

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 a new, newarray, or anewarray bytecode, 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 isn't, then the heapAlloc pointer is bumped so that it maps to the beginning of L1 cache, and the space in between is marked as used (so that the GC does not get confused). 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.

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 we are using an AMD Opteron (Shanghai) processor, which has 64 KB of L1 data cache, and is 2-way set associative with a 64-byte line. 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, this means that we can think of the TLH as being made up of 64 KB/2 = 32 KB chunks, and we want to allocate objects in the last 25% of them. The location where a physical address maps to the cache is determined by its address's 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 (2^15 = 32 KB). This is done in the compiled assembly code using bitwise arithmetic. N, as well as the cache size and associativity are all constants that can be changed in the prototype.

One might ask why we would allocate *all* objects in this way. The idea is for this to be used with generational GC that is unaware of CLR. The generational GC allocates new objects in the area of the heap known as the *nursery*. As the nursery gets filled up by allocated objects, when it reaches half of its capacity, a collection is made. The GC 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.

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, this can be done as before, by incrementing the tlhAlloc pointer. The disadvantage of this scheme is that objects might occupy non-reserved sections. Other limitations include having only one reserved section and having no control over what objects to reserve (the heap will fill up quickly if the reserved section is small, because of constant bumping of tlhAlloc).

4.2 "Strong" CLR – Prototype 2

We developed a second prototype 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. 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 are 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, because it will not overshoot tlhReserved.) Figure 3 shows the main steps that occur when a reserved object gets allocated. Again, whenever a pointer is bumped inside the TLH, to prevent having a hole of unallocated space in the middle of the TLH, which confuses the GC, this space is marked as used and will be reclaimed at the next GC.

When the JIT is compiling a method that allocates an object, it decides whether to generate code for reserved or normal allocation; it decides this based on the object name (or if it is an array, based on the name of the method that contains it). Even though Figure 3 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 incre-

mented (after some untaken conditional branches), just as tlhAlloc is incremented in the non-reserved, conventional case.

4.3 "Strong" CLR/non-CLR – Prototype 3

The second prototype guaranteed that a reserved object will be in a reserved section, but it did not guarantee that the non-reserved object will *not* be in a reserved section. In the third prototype, as an extension to the second prototype, we allocate non-reserved objects *only* to the non-reserved regions. Reserved allocations are done identically as in prototype 2. Also like in the second prototype, we can still 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.

5 Preliminary Results

We present a basic CLR proof of concept in C to show that it works. Then, we do the same with a custom benchmark in Java. Finally, we investigate CLR in relation to SPECjvm2008 and SPECjbb2005. Rather than publishing actual scores (lower is better), we offer a comparative analysis (CLR vs. no CLR). All tests were done on an AMD Opteron (Shanghai) processor running Microsoft® Windows® Server 2003. Its L1 cache is 64 KBytes, and is 2-way set associative, with a 64-byte line size. Although this processor has 8 cores, only 4 were used, to prevent non-uniform memory access (NUMA) variations. The AMD Code Analyst was used to obtain data from hardware counters. The *miss rate* is the number of memory access instructions that caused an L1 cache miss, out of the total number of executed instructions. The *miss ratio* is the number of memory access instructions that caused an L1 cache miss, out of the total number of memory access instructions. The *evicted rate* is defined as the number of evicted L1 cache lines divided by the number of total executed instructions.

5.1 Proof of Concept

We have created a proof-of-concept C program that mimics what prototype 2 would do on a custom Java benchmark.

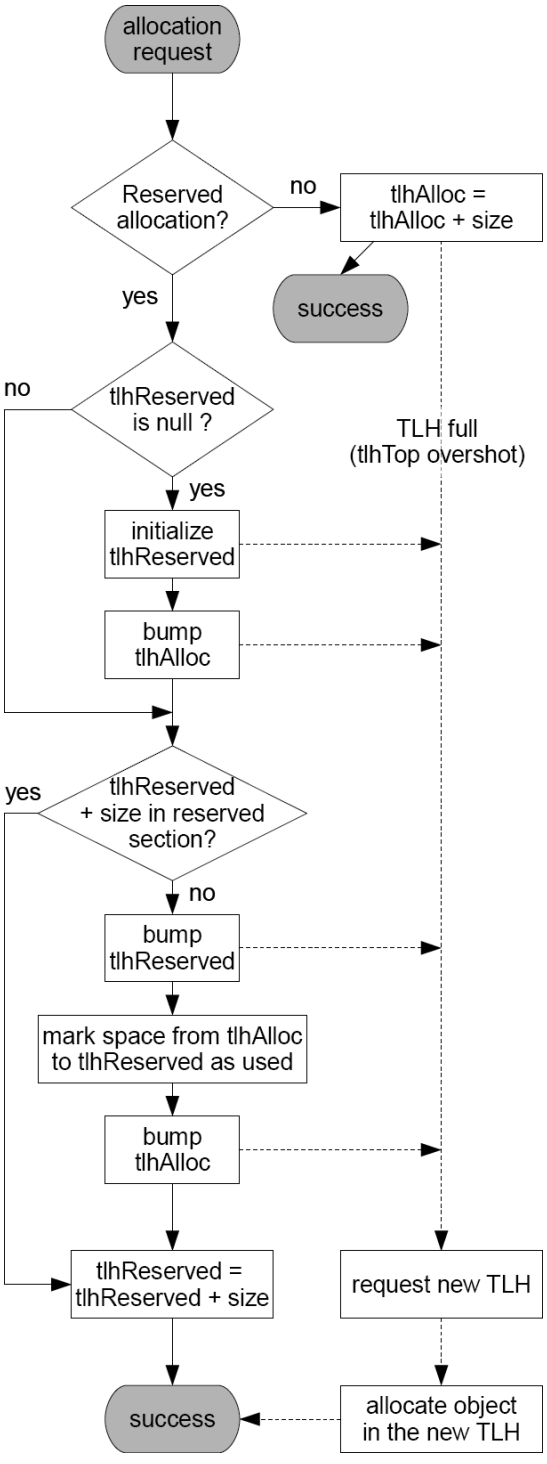

Figure 3: Processing a reserved allocation in the strong CLR prototype

The following steps were performed in the program to emulate non-CLR allocation:

- Allocate using malloc a continuous area of memory (192 KB or 6 chunks), and assign it an integer pointer.
- 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, because a single cache line holds 16 integers (of 4 bytes each).
- 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.

The second scenario is trying to mimic the CLR scheme from prototype 2 where 25% of the cache lines are reserved, and where the first array has been reserved (but not the second one):

- Allocate 192 KB 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 measure the time.

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 space in the chunks (chunks 3, 4, 5 and 6 are only 25% filled with objects), this will mean that half of total accesses will be hits, and the other half misses.

Test	Time (s)	Miss rate	Miss ratio
no CLR	5.97 (100%)	0.209	0.7368
CLR (25%)	3.72 (62%)	0.114	0.4232

Table 1: Proof-of-concept cache profile.

Table 1 shows the results of the tests. From the results, we can see that CLR can offer a significant improvement in performance. Due to overhead, the miss ratio and the miss rate might not have halved, but the results are close to this goal.

The C code was compiled using Microsoft's Optimizing Compiler Version 14.00, with the /O2 option (maximize speed).

5.2 Custom Benchmarks

A microbenchmark similar to the one from the proof of concept was developed in Java to test out the Java prototype and confirm that CLR can provide a benefit. This is what was set up in the benchmark:

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

This benchmark gives us the flexibility to discover whether CLR works in the expected way in the Java environment, according to the physical

cache structure. As before, we reserve 25% of the cache, and prototype 2 was used (strong CLR reservation). Table 2 summarizes the results.

	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.

Table 2: Custom benchmark performance

From Table 2, you can see all the variants of the benchmark that were performed. The results confirmed what was expected. When we had only one linked list, CLR slowed things down significantly. (If we look at the R case, the test took 2 times longer to run with CLR when 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 because there is improvement only when we allocate 3 or more linked lists. (The two last ones 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. (Execution times were 10.55 seconds without CLR, and 6.48 seconds with CLR, which is 61% of the non-CLR case, compared with 62% in the proof-of-concept C program.) Profiling of the NNR case (Table 3) shows that the performance improvement does indeed come from a reduction in cache misses.

	Executed Instructions (% of total ticks)	Miss Rate	Miss Ratio
no CLR	68943 (76.4%)	0.136	0.152
CLR (25%)	149884 (79.9%)	0.066	0.097

Table 3: Cache profile of the RNN read/write run on the custom benchmark

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.

5.3 SPECjvm2008

SPECjvm2008 [25] is a suite of benchmarks that was designed to measure performance of the Java Runtime Environment (JRE). We have used prototype 1 so that all objects are allocated to the reserved areas, in the hope of identifying any improvement that CLR can bring. Unfortunately, there were no results where CLR provided a benefit. Reducing the size of the reserved region decreased performance, because TLHs would fill up more quickly.

Next, we reserves specific objects, using prototype 2, in an attempt to perform more targeted reservations. We have reserved the following objects: Strings, StringBuffers, StringBuilders, and BigDecimals. The char[] array in String, StringBuffer, and StringBuilder instances was also allocated to a reserved section (instead of allocating just the container objects). Once again, CLR only hurt performance.

5.4 SPECjbb2005

SPECjbb2005 [25] is a benchmark for evaluating server-side Java. Similar tests were performed as in SPECjvm2008 with all three prototypes, but 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 a real-life benchmark. In an attempt to find CLR opportunities, we have identified a method, *populateXML*, that gets

called often during SPECjbb2008. It goes through a 2D char array (of size char[24][80]), and for each row in the array, it creates a String object. Every time populateXML is called, 24 String objects are created. Each of these String constructors has to create its own char[] array, of length 80. It does that by allocating a new array and *copying* the contents from the 2D array to the new array. Therefore, every time populateXML allocates a String, a new array has to be allocated and brought into the cache. These Strings are not zero-lifetime objects, but they do consistently die as they are used to fill up a buffer that has a limited capacity. Our diagnostic measurements indicate that for one warehouse (one thread) in SPECjbb2005, 2% of the time is spent on the String allocation, 7% on writing to the new char array in the String constructor and 2% on reading the contents of the 2D array.

	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.

Table 4: Cache profiles of SPECjbb2005 when reserving Strings in populateXML

Table 4 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 with the "no CLR" run, the score with CLR was worse, but the cache performance didn't seem to improve much (if any). 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. We suspect that the reason for a similar cache profile between the "CLR" and "no CLR" runs is as follows. One char in Java is 2 bytes, so an 80 character array is 160 bytes. Every time populateXML is called, 24 of those character arrays are allocated, for a total of 3840 bytes. If each cache line is 64 bytes, that means that every time that populateXML is called, 60 lines worth of reserved objects is allocated (and the whole cache contains 1024 lines, with an associativity of 2). These objects will therefore evict 60 cache lines. When CLR is disabled, these 60 lines will be random. When 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 most probably 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 were called more often, or, if the array were bigger than 60 lines.

	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.

Table 5: Cache profiles of SPECjbb2005 when reserving larger Strings in populateXML

When the prototype was tested with a bigger array, it was clear that the cache misses had decreased, as Table 5 shows. Instead of being 24 by 80, the size of the 2D character array was artificially changed to be 120 by 320. However, there was still no improvement in the overall score. The reason for this is a large GC overhead that is incurred.

To try to investigate and minimize the GC overhead, we decided to try different GC policies (generational and conventional), as well as different sizes of the heap. Table 6 shows the results. We can reach two conclusions. First, increasing the heap size from 1.77 GB to 3.54 GB 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 GC suffers from less overhead than optthruput GC. 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.) 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.

To eliminate the overhead of GC, we shift our focus from reserving objects that are *fre-*

quently 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. In Section 3.1, we have identified them under the "Objects accessed in infrequent bursts" heading.

Heap Size		Proportion of time spent in: GC, application			
		optthruput		generational	
1.77 GB	no CLR	14%	83%	5.4%	91%
	CLR	36%	60%	17%	80%
3.54 GB	no CLR	8.0%	89%	3.3%	93%
	CLR	24%	72%	10%	87%

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

All tests with CLR showed a significant reduction in the miss rate/ratio, similar to Table 5.

Table 6: Investigating the effect of different GC policies and heap sizes on CLR

Unfortunately, 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. Each time it allocates a String object in the normal benchmark, we make it traverse a linked list. We used the same linked list 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 the linked list 24 times.
Table 7 shows that CLR does indeed yield a significant benefit (around 30%) in this specialized benchmark. The challenge is to identify these objects automatically. This improvement is only seen with the optthruput GC policy. With generational GC, 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.

	Score	Miss Rate	Miss Ratio	Evicted Rate
no CLR	100%	0.0369	0.0788	0.278
CLR 25%	130%	0.0196	0.0405	0.244

Measured with 4 warehouses for 4 minutes.

Table 7: Cache profiles and scores of SPECjbb2005 where populateXML traverses a single linked list instead of allocating Strings

6 Conclusions and Future Work

This paper introduces a novel idea of cache line reservation (CLR) for allocating objects. Selected objects are allocated to a reserved section of L1 cache with the aim to reduce cache misses. A proof of concept of CLR was presented in the form of a C program.

Three prototypes of CLR were developed in the J9 JVM and the Testarossa JIT compiler. They can serve as starting points for further development. Some custom Java benchmarks do show the potential benefits of CLR, but the current implementation has limitations. The garbage collection has significantly increased, as well as TLH requests. The results seen in custom benchmarks indicate that CLR should concentrate on reserving long-lived objects that are allocated only once but accessed often because this will eliminate the GC overhead.

Future work should focus on trying to identify objects suitable for reserved allocation, first manually, and eventually automatically, during runtime. In addition, new GC schemes could be developed that would be CLR-aware and CLR-friendly.

Traditionally, CPU caches were managed exclusively by the hardware. In the future, as the sizes of caches increase, it will become worthwhile for compilers to start using the internals of the cache for advanced optimizations. Cache line reservation is a simple idea that can offer benefits, but making it work on a consistent basis will be a challenge that is presented to the compiler community.

Acknowledgements

This research was funded in part by the Natural Sciences and Engineering Research Council.

Trademarks

About the Authors

Ivan Bilicki is pursuing an MEng with McGill University in the Department of Electrical and Computer Engineering. After working as a co-op student with the IBM JIT compiler team in summer 2008, he continued his research as an IBM CAS (Centre for Advanced Studies) student. He received his BEng in honours electrical engineering from McGill University in 2007.

Vijay Sundaresan, Daryl Maier, and Nikola Grčevski are currently working for the IBM Testarossa JIT compiler optimizer team at the IBM Toronto Lab. Their focus is improving the performance of the JIT compiler on various computer architectures.

Željko Žilic is an associate professor in the Department of Electrical and Computer Engineering at McGill University. He is researching various aspects of system design, test, and verification. He has a MSc and a PhD in electrical and computer engineering from the University of Toronto.

References

[1] J.L. Hennessy and D.A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann, 2006.

[2] J. Kim and Y. Hsu, "Memory system behavior of Java programs: methodology and analysis," *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.

[3] H. Inoue, D. Stefanovic, and S. Forrest, "On the prediction of Java object lifetimes," *Computers, IEEE Transactions on*, vol. 55, 2006, pp. 880-892.

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

[5] T. Sheu, Y. Shieh, and W. Lin, "The selection of optimal cache lines for microprocessor-based controllers," *Proceedings of the 23rd annual workshop and symposium on Microprogramming and microarchitecture*, Orlando, Florida, United States: IEEE Computer Society Press, 1990, pp. 183-192.

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

[7] A. Janapsatya, A. Ignjatović, and S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," *Proceedings of the 2006 conference on Asia South Pacific design automation*, Yokohama, Japan: IEEE Press, 2006, pp. 796-801.

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

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

[10] R. Subramanian, Y. Smaragdakis, and G.H. Loh, "Adaptive Caches: Effective Shaping of Cache Behavior to Workloads," *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2006, pp. 385-396.

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

[12] C. Lebsack and J. Chang, "Using scratchpad to exploit object locality in Java," *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. 381-386.

[13] K.F. Chong, C.Y. Ho, and A.S. Fong, "Pretenuring in Java by Object Lifetime and Reference Density Using Scratch-Pad Memory," *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, 2007, pp. 205-212.

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

[15] J. Kalamatianos, A. Khalafi, D. Kaeli, and W. Meleis, "Analysis of temporal-based program behavior for improved instruction cache performance," *Computers, IEEE Transactions on*, vol. 48, 1999, pp. 168-175.

[16] D. Genius, "Handling Cross Interferences by Cyclic Cache Line Coloring," *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 1998, p. 112.

[17] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *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.

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

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

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

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

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

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

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

[25] "SPEC - Standard Performance Evaluation Corporation", Internet:http://www.spec.org/, [May 2009]