

Compiler Support for Reference Tracking in a type-safe DSM

Ralph Goeckelmann, Stefan Frenz, Michael Schoettner, Peter Schulthess

Distributed Systems Laboratory, Ulm University o-27, D-89069 Ulm
schulthess@informatik.uni-ulm.de

Abstract. The efficiency of language implementations is heavily influenced by the selected strategy for allocation and reclaim of memory. Memory allocation in a distributed shared memory (DSM) cluster poses additional challenges. Designing the DSM as a distributed heap is natural and relieves the application programmer from the burden of memory management. Garbage collection is incremental and refrains from repeatedly marking, sweeping and writing to the distributed memory. Relocation of objects is implemented to reduce memory fragmentation and to resolve false-sharing conflicts. Reference tracking and a type-safe language are essential for garbage collection and object relocation. In this paper we present a novel data structure which we call “backpacks” used to efficiently keep track of global references in our language-based DSM. We also show how our home grown Java Compiler supports reference tracking and garbage collection by generating bi-directional runtime structures.

1 Introduction

Safe memory allocation and reclaim is based on the concept of strong typing provided by the programming language. This type safety is typically restricted to a single machine and its main memory. Once we leave the single station environment and the main memory, middleware packages take over and we lose the benefits of the safe type system. In the Plurix operating system (OS) we extend the typing introduced by the Java programming language to a cluster of workstations and provide type-safe memory-management in a distributed shared memory (=DSM) [1]. Language based OS development has been successfully demonstrated in systems like Oberon [2].

DSM systems introduced by Keedy and Li offer the abstraction of a single address space to all stations of a cluster [3], [4]. Traditional DSM environments are built on top of Unix or Microsoft Windows systems and do not exploit type-safe languages but use C with its known drawbacks. Furthermore, they offer only coarse grain memory allocation primitives and it is the burden of each application to implement allocation and de-allocation functions [5]. This approach is error-prone, time consuming, and typically causes performance penalties. We avoid these disadvantages by implementing our Java-based DSM as a global heap optimized to achieve high performance. Language objects (classes, interfaces, instances, and arrays) are simply allocated, manipulated and eventually recollected in this global heap. The latter is the responsibility of our distributed garbage collection.

Modern programming languages (e.g. Oberon, C#, Java) typically offer automatic garbage collection. As the programmer may no longer explicitly free memory, dangling references and memory leaks are avoided. In a DSM environment it is not a good idea for a distributed garbage collection (GC) scheme to mark or to colour the heap blocks during a traversal because collision probability grows with the number of written objects. The “backpack”-scheme proposed in this article is designed to support efficiently distributed garbage collection for a DSM heap avoiding these drawbacks. The “backpack”-scheme is also the foundation for a dynamic object relocation facility used to resolve false-sharing situations and to compact fragmented parts of the heap.

Our Plurix Java Compiler (PJC) is used to develop the OS and the applications. It directly translates Java source texts into Intel machine instructions and is an integral part of the OS [6]. Native code generation is mandatory for OS and driver development in Java. The PJC generates bi-directional runtime structures and supports reference tracking to simplify garbage collection and object relocation.

Our paper is composed of five sections starting with a description of bi-directional runtime structures. In section three we present the backpacks - our novel reference tracking scheme. In section four we discuss memory allocation, garbage collection and object relocation. Finally, we present selected performance measurement data.

2 Bi-Directional Runtime Structures

We decided to implement bi-directional heap blocks (Fig. 1) to simplify the garbage collection process, to support the relocation of objects and to facilitate a periodic heap consistency check. A reference will always point to an object header in the middle of the block. To the left of the header we find references to other objects, including code segments and class descriptors. The first reference on the left side is always a pointer to a class descriptor defining the type of each memory block. To the right we find scalars and scalar array elements.

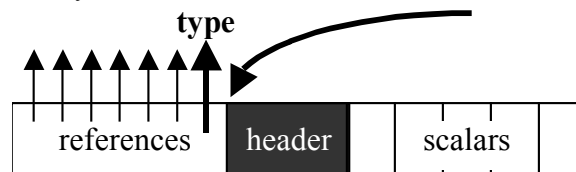


Fig. 1. Bi-directional layout scheme of heap blocks

Our Plurix Java Compiler generates all runtime structures according to this bi-directional scheme. Numerous variants of runtime structure memory layouts have been discussed for object-oriented languages especially to support multiple inheritance. To the best of our knowledge no other implementation of an object-oriented language has established a bi-directional layout for separating references from scalars. Of course bi-directional layouts are well known and have been used for other purposes.

For example Myers used it to implement multiple sub-typing in the Theta language [7]. Using a bi-directional memory layout he achieves class/super-class compatibility by separating fields from the dispatch header. The field part, however, may still contain references to other instances and differs from our approach.

Language implementations with automatic garbage collection often require voluminous offset tables to distinguish between pointer references and scalar variables whose value might look like a memory address. With bi-directional heap blocks the garbage collection only needs to scan the reference portion of the heap block and all entries are known to be references.

In principle each Java object is allocated as a separate heap-block which can reference further objects (including arrays and strings). A separate block is created even if the size of the object is known at compile time and the compiler would be in a position to embed a referenced object directly into the original object. We thus risk obtaining a heap with an unnecessary amount of fragmentation and with many small blocks which are expensive to manage and to collect.

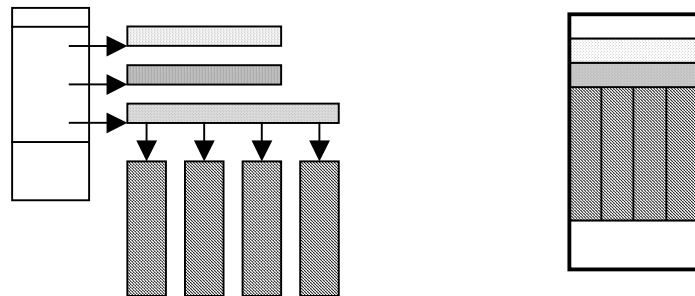


Fig. 2. Java compatible versus Plurix monolithic memory layout

This object fragmentation in the heap is particularly disturbing because the compiler must set up class descriptors. These contain scalars and arrays (e.g. for the method jump table) which in real Java would lead to complex linked list structures (Fig. 2). The compiler creates a linearized and monolithic class descriptor by bypassing the type rules. A detailed description of our runtime structures and language extensions can be found in [8].

3 Reference Tracking

To support reference tracking the Plurix Java Compiler generates a runtime call for each pointer assignment to a heap reference. This method is very short and is only used to update the bookkeeping of references. To reduce the overhead caused by the call during pointer assignment we only track heap references. Pointers residing on the stack must not be considered because garbage collection and object relocation is only executed if the stack is empty (see section 4). This is not a heavy restriction because we do not implement preemptive multithreading but transactional processing with a cooperative multitasking scheme.

3.1 Backchain – our old solution

Considerable thinking effort has been spent on the run-time structures in the distributed heap of our OS prototype. An early implementation of our heap organization used a *backchain* scheme (Fig. 3) where all references to an object were linked into a serial list starting at the object itself.

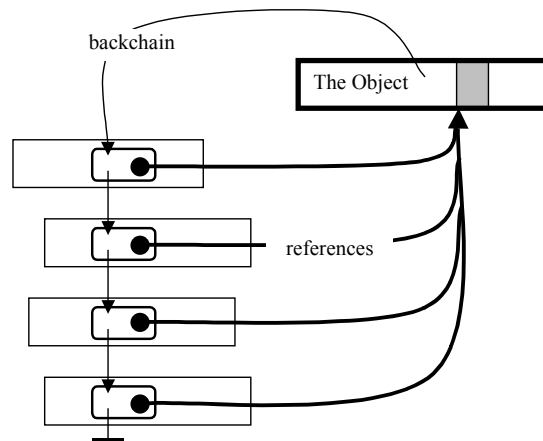


Fig. 3. Serial backchain linking all references to an object

Since each reference was 32 bits wide the full container size for a pointer was increased to 64 bits to include the backchain link. The chain conveniently supports relocation of objects and incremental garbage collection. Any object with an empty backchain is garbage and can be reclaimed. When an object is reclaimed it might itself reference some objects and thus be linked to their Backchain. It is now necessary to unlink all freed reference variables from their respective chains. This requires an expensive linear search in a chained list, and a lot of memory pages might be fetched over the network if not present locally. Furthermore, garbage cycles are not recognized and might require a separate mark and sweep procedure [9].

The backchain requires updating whenever an assignment or de-assignment to a heap reference occurs. This updating of the backchain could become costly and lead to unpredictable invalidation patterns in the distributed heap (see 4.1). New references to an object are always inserted at the head of the list and the object itself is invalidated. In case of a de-assignment any object within the backchain may be invalidated because its pointer need to be updated to reference the successor of the pointer to be detached.

Statistics showed that backlinks were mostly empty and when proceeding from a 32 bit address space to 64 bit addresses in some distant future the waste of memory will become excessive. The backchain concept was eventually discarded in favor of the backpack scheme described in the next paragraph.

3.2 Backpacks – the new solution

Using so-called *backpacks* instead of the backchain of the early implementation proved to be superior with respect to the invalidation patterns and to the storage requirements. The scheme is illustrated in Fig. 4.

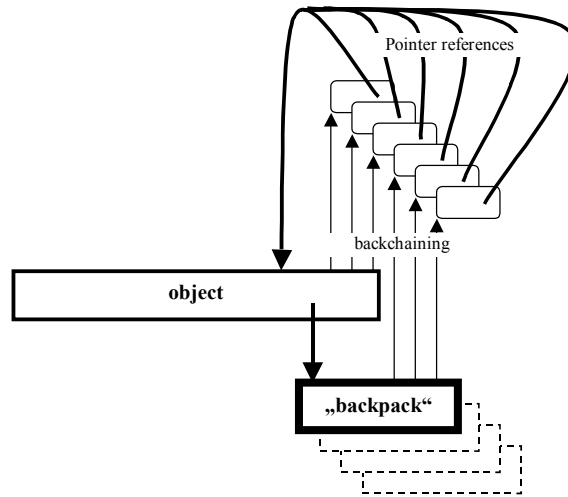


Fig. 4. Backpacks tracking the references to an object

The references to an object are tracked in a separate data structure called *backpack* which is an object by itself and contains the memory addresses of the respective pointers. The first three references are tracked from direct back-links which are located in the object itself. A backpack is thus only created if there are more than 3 global references to an object.

References to widely used systems objects, e.g. the class descriptor of “java.lang.string”, are not tracked and do not trigger the creation of backpacks. In fact these objects are allocated in separate memory pool to be specified at compile time or at run time. In actuality less than 10% of all objects require the creation of a backpack, see section 5. The management of backpacks adds further complexity to the memory manager but this complexity is rewarded by superior DSM performance. Updates to the bookkeeping of references require less network traffic due to the better access locality. Furthermore unpredictable invalidations caused by backchain maintenance (see 3.1) do no longer occur. Finally, the backchain approach requires double pointer size - the backpack approach instead adds overhead only when necessary. This backpack scheme has been the subject of a recent patent application.

4 Heap Management

Programming languages typically implement two separate strategies of memory allocation: stack allocation and heap allocation. Early Pascal implementations simply marked the top of the heap before proceeding with further allocation. Later all allocations beyond the mark were released in one step – thus simulating a second stack.

More sophisticated schemes are able to individually allocate and free single objects – either explicitly or by implicit automatic garbage collection. Buddy systems split and combine memory blocks of size $k \cdot 2^n$ to reduce heap fragmentation [10]. Separate free-storage chains may be maintained for different container size to reduce the search for a suitable container. The search for a chunk of memory may be “first fit”, “best fit” etc. However, memory allocation in a DSM poses additional challenges.

4.1 Distributed Memory Allocation

Modifications within a node of the DSM must be propagated and coordinated. Most DSM systems implement the write-invalidate coherence protocol to invalidate all read-only copies of a memory page before a node is allowed to write it. Furthermore, memory consistency models define when write operations become visible for other stations. In a single station model any write to main memory is immediately visible for all subsequent read operations. This is known as strict consistency – the most comfortable programming model – but the most expensive in a distributed system due to excessive network latencies. As a consequence DSM designers propose weaker models delaying the propagation of write operations and thus improving memory performance [11].

Implementing a distributed heap structure within a DSM obviously requires a strict memory consistency model to avoid inconsistencies within the memory manager. Clearly it is difficult to implement strict consistency in a DSM but the Plurix project introduces *transactional consistency* solving this problem. Any command in our system is executed as a restartable transaction and concurrent TAs are serialized using an optimistic synchronization scheme [12].

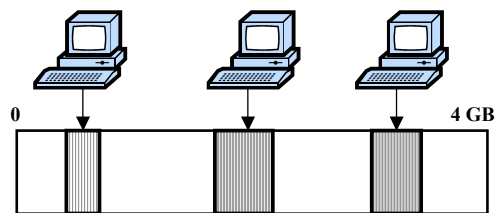


Fig. 5. Distinct heap entries per node

Memory allocation in a DSM should try to leave adjacent heap blocks unmodified to avoid potential collisions with blocks used by other nodes. Using different entry points in the heap will cluster objects allocations of each node and improves locality (Fig. 5). Special memory pools must be used to protect vital components of the system from invalidation, e.g. kernel. More details about memory pools in [13].

4.2 Garbage Collection using Backpacks

Reclaiming memory is the task of our distributed garbage collector (GC). The bi-directional memory layout of the runtime structures lets the GC identify pointers without additional overhead. The addresses of object references are provided by the backpacks without additional calculations. If there are no references to an object it is obviously garbage and can be collected. Using the backpacks and backlinks the GC can easily detect unreferenced objects.

Each object not reachable from the global root of the cluster-wide naming service is garbage. Detecting cyclic garbage efficiently in our DSM environment is still under study. The Plurix system saves checkpoints in short intervals on disk to survive node failures. We are considering running an off-line GC on the last DSM snapshot to identify cyclic garbage. Potential candidates to break a cycle can be searched and a hint can be posted to the running cluster.

Incremental GC is periodically executed when the station is idle (the stack is empty) or running out of virtual memory. The latter situation only occurs if the total cluster runs out of memory and will virtually disappear with the advent of 64 bit processors. Executing the GC only on an empty stack allows the reference tracking to ignore assignments to pointers residing in the stack. This is not a critical restriction as we have no multithreading per node but a transactional processing. However, multiple nodes are allowed to concurrently execute incremental garbage collectors.

Collisions between a GC and an overlapping application transaction executing on another node are detected and the GC will then abort tacitly. The object(s) that caused the collision were inspected by the GC but obviously are still in use. We plan to use such collision information as an input for the next restart of the GC.

4.3 Object Relocation

Relocation of objects is simplified because moving an object by a certain offset amount can be compensated by adjusting all references in the backpack. The transactional processing allows the object relocater to be executed concurrently with other transactions on other nodes - any collisions are detected and resolved.

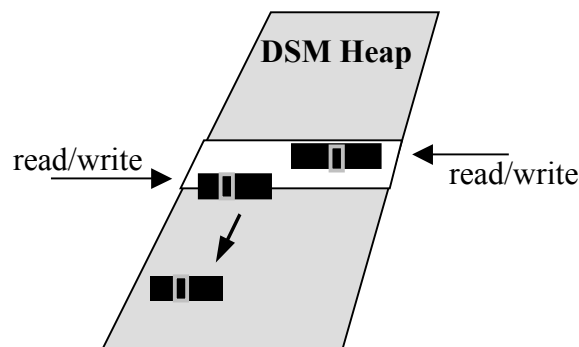


Fig. 6. Relocation of an object false-sharing a single memory page

Relocation may be necessary to reduce heap fragmentation or to control false-sharing situations. False-sharing causes significant performance penalties in page-based DSM systems because of heavy page trashing [14]. If two objects reside on a memory page and within a certain time interval each of the object is accessed by a different node access conflicts may arise which are semantically unnecessary. Unfortunately, false-sharing is a time dependent phenomenon and it may turn into true-sharing in some later time interval. Our object relocater is able to resolve such situations by relocating involved objects to other free memory pages, see Fig. 6.

Of course relocating objects tends to store only one single object per page destroying locality benefits of the page granularity. Currently we are developing policies to cluster scattered objects using the relocation facility to improve locality by true-sharing.

5 Evaluation and Conclusion

The table on the next page shows preliminary statistics of the characteristics of the distributed heap (object types and backpacks) and the start-up time of single nodes. The computers are booted in numerical order. System objects must not be invalidated because they are essential for system operation (e.g. the page-fault handler) and thus require special treatment by the memory management. Double numbers (*normal count*/*system count*) appearing in table 1 indicate normal and system objects. Column 4 shows the values for three nodes running the oberon like GUI.

The first section shows the heap content after start-up of the system consisting of ~40.000 lines of OS and driver code. In the second section the backpacks statistics are presented for the respective heap objects. Most of the objects do not require a backpack as they are referenced by less than three instances only. Joining nodes allocate new instances with possible new backpacks but classes and methods are shared through the DSM. The following section shows the usage of backlinks (three available per object). Most of the objects need only a single backlink. The inheritance section lists the inheritance depth of classes. Dynamic methods are replicated in method jump tables of subclasses and the code-segments are shared by subclasses. As a consequence code-segments may be referenced by more than three classes and may then require a backpack. Fortunately, most of the class inheritance hierarchies have less than three levels. The final part shows startup time with backpacks and an OS version compiled without bookkeeping of references. Startup time of Node 1 is slower because it creates the initial DSM heap. Node 2 and 3 only join and fetch desired objects from the already running heap.

These figures support our hypothesis that the backpack scheme can be implemented without excessive penalties in memory usage and execution time. Together with bi-directional runtime structures generated by our Java compiler it is an elegant foundation for incremental garbage collection and object relocation. The latter is essential for heap compaction and resolution of false-sharing situations.

Table 1. Measurements for system startup.

	Node 1 (P4 2,54 Ghz)	Node 2 (Athlon XP 2,0)	Node 3 (Athlon XP 2,2)	Running GUI
Objects				
- total	7339	7673	7987	8556
- instances	2032 / 253	2239 / 263	2444 / 273	2727 / 273
- methods	1291 / 582			
- classes	170 / 58			
- backpacks	1009 / 0	1071 / 0	1133 / 0	1270 / 0
- arrays	1710 / 234	1765 / 234	1802 / 234	1951 / 234
Backpacks				
- total	1009	1071	1133	1270
- instances	550 / 253	583 / 263	599 / 273	745 / 273
- methods	136 / 0			
- classes	184 / 0			
- arrays	139	168	197	201
Backlinks				
only 1	4556 / 198	4636 / 205	4715 / 212	5097 / 212
two	379 / 0	339 / 0	338 / 0	362 / 0
three	105 / 0	137 / 0	98 / 0	123 / 0
>three	276 / 1	316 / 1	387 / 1	398 / 1
Inheritance				
only 1	4580 / 535	4865 / 537	5166 / 541	5655 / 541
two	1330 / 588	1357 / 591	1384 / 594	1392 / 594
three	19 / 6	20 / 9	21 / 12	57 / 12
>three	283 / 0	284 / 0	285 / 0	287 / 0
Startup time				
backpacks	803 ms	350 ms	235 ms	-
no backp.	794 ms	350 ms	235 ms	-

6 References

- [1] Plurix Homepage: www.plurix.de
- [2] N. Wirth and J. Gutknecht, *Project Oberon - The Design of an Operating System and Compiler*, Addison-Wesley, 1992.
- [3] J.L. Keedy and D.A. Abramson, "Implementing a large virtual memory in a Distributed Computing", *Proceedings of the Hawaii International Conference on System*, Hawaii, USA, 1985.
- [4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", *Proceedings of the International Conference on Parallel Processing*, 1988.
- [5] M. Pizka and C. Rehn, "Heaps and Stacks in Distributed Shared Memory", *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, Ft. Lauderdale, Florida, USA, 2002.
- [6] M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999.
- [7] A.C. Myers, "Bidirectional Object Layout for Separate Compilation", *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Texas, USA, 1995.
- [8] M. Schoettner, *Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher*, Dissertation, Universität Ulm, 2002.
- [9] F. Le Fessant, "Detecting distributed cycles of garbage in large-scale systems", *Principles of Distributed Computing*, Rhodes Island, August 2001.
- [10] J. L. Peterson and T. A. Norman, "Buddy systems", *Communications of the ACM*, 20(6), 421 - 431, 1977.
- [11] D. Mosberger, "Memory Consistency Models", *ACM Operating Systems Review*, 27(1), 18-26, January 1993.
- [12] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, "Optimistic Synchronization and Transactional Consistency", *Proceedings of the Workshop on Distributed Shared Memory on Clusters*, IEEE International Symposium on Cluster Computing and the Grid, Berlin, Germany, 2002.
- [13] R. Goeckelmann, M. Schoettner, M. Wende, T. Bindhammer, and P. Schulthess, "Bootstrapping and Startup of an object-oriented Operating System", *European Conference on Object-Oriented Programming - Workshop on Object-Oriented and Operating Systems*, Malaga, Spain, 2002.
- [14] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance", *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, San Diego, USA, 1993.