Java objects without the header

Kris Venstermans, Lieven Eeckhout

Supervisor(s): Koen De Bosschere

Abstract—**By a recent trend, 64-bit machines are invading the end-user market. One major rumor keeps people from switching over: a 64-bit machine uses more memory then its 32-bit counterpart for the same job. For a 64-bit Virtual Machine, we tackle 2 reasons that nurish this rumor. For a given application we identify the key object types. Instances of these key types get allocated in a type specific high 64-bit virtual address range. Then for these key 64-bit objects, we remove the enlarged object header and we reduce inter-object alignment. Information previously stored into the header, is now no longer uniformly accessible for all objects. The extra overhead to obtain this information is won back again for most benchmarks by the better memory layout in the heap. As a result we obtain applications that run equally fast, but allocate less memory.**

Keywords—**Java, memory management, 64-bit, Implicit typing**

I. Introduction

Java is very popular on a various computing systems, ranging from embedded devices to high-end servers. There are several reasons for the popularity of Java: its platform-independence as it relies on virtual machine (VM) technology, its object-oriented programming paradigm, its reliance on automatic memory management, etc. All of these factors contribute to the productivity enhancement of software development using Java.

Memory performance is an important design issue for contemporary systems given the ever increasing memory gap. This paper proposes a space-efficient Java object model for reducing the memory consumption of 64-bit Java virtual machines. In total we reduce the header size for an object from 16 bytes to only 4 bytes. This results in a reduction of memory consumption by on average 15% (and up to 38% for some benchmarks). In terms of performance, we get statistically significant performance speedups for several benchmarks, up to 23%.

II. IMT concept

The idea behind the Implicit Typing Method (ITM) is to encode the object's type in the object's virtual address. As such, memory segments are created in wich all allocated objects are of the same given type. The Type Information Block (TIB) can be computed from the object's virtual address, instead of through a load instruction. For ITM-enabled objects, the TIB can be removed from the object's header which reduces the number of allocated bytes per object.

Besides the elimination of the TIB from an object's header, we also compact the status word by removing the forwarding pointer from it. A status word typically contains hash, lock an gc information. Whenever a forwarding pointer is needed, we overwrite the remaining 4 byte status field as well as the first 4 bytes of the object's data.

Unlike previous work on implicit typing [Baco02], [Stee97], [Hans80], we apply ITM to a selected number of frequently allocated and/or long-lived object types. This is done in order to limit the amount of memory fragmentation. In order to select an object type to fall under ITM, the object type needs to apply to one of the following criteria.

In our first criterion we make a selection of object types of which a sufficient amount of objects is allocated. Through a profiling run, we collect how many object allocations are done for each object type, and what the object size is for each object type. Then we compute for each type the total number of allocated header bytes, and we compute the procentual volume of these header bytes in relation to the total number of allocated bytes. We then select object types for which this procentual volume exceeds a given *memory reduction threshold (MRT)*.

In our second criterion we limit the scope to long-lived objects because they are likely to remain in the heap for a fairly long period of time. For those objects that survive a garbage collection, we again compute the procentual volume of the header bytes in relation to the total number of allocated bytes. We then retain object types for which this procentual volume exceeds the *long-lived memory reduction threshold (LLMRT)*.

III. IMT overhead

Our ITM-enabled implementation clearly has an overhead compared to a traditional VM implementation. In an ITM-aware VM implementation, we make a distinction between a ITM-enabled object and a ITM-disabled object. This distinction often can be only made at runtime, leading to a runtime conditional branch. The single most impediment to a more efficient implementation is the branch that is conditional on whether the object is ITM-enabled or ITM-disabled. Unfortunately, in our PowerPC implementation we could not remove this conditional branch through predication. Nevertheless, this could be a viable solution on other ISAs, e.g. the `cmov` instruction (Alpha) , IA-64 ISA (full predication).

In order to limit the overhead of the conditional branch we try to limit the frequency of going through this relatively slow TIB access path. This is done by marking the class tree with the ITM-enabled object types. A subtree is marked in case all types in this subtree are ITM-disabled, in which case the TIB can be loaded in a non ITM-aware implementation.

Second form of overhead is in the memory allocator.We now need to keep track of multiple allocation pointers that point to the free space in all the object type specific segments. The selection of an individual allocation pointer requires an extra indirection for ITM-enabled object types.

And as third, ITM has an impact on garbage collection. Dif-

ELIS department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium. E-mail: {kvenster,leeckhou}@elis.UGent.be

Kris Venstermans is supported by a BOF grant from Ghent University. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (FWO - Vlaanderen). This research is also sponsored by the Promotion of Innovation by Science and Technology in Flanders (IWT) and the HiPEAC network

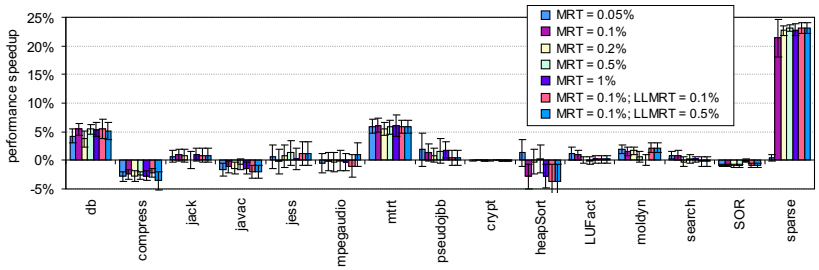

Fig. 1. Performance speedups (with 95% confidence intervals) as a function of MRT and LLMRT.

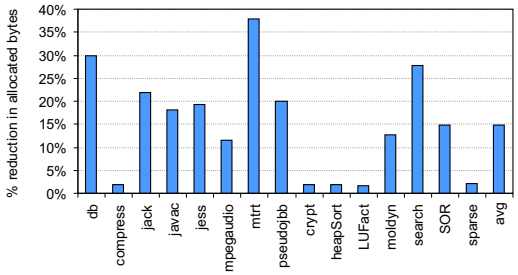

Fig. 2. Reduction in the number of allocated bytes for MRT = LLMRT = 0.1%.

ferent actions may be undertaken depending on whether an object resides specific memory regions. Since our IMT-enabled VM now has more distinct memory regions, this incurs additional overhead to determine which region an object lays in.

IV. Results

Figure 2 shows the reduction in allocated bytes, obtained from a cross-validation setup, *i.e.*, we use profile inputs for selecting the ITM-enabled types, and we use reference inputs for reporting performance speedups. We observe an average reduction in allocated bytes of 15%. For some benchmarks we even observe a reduction in allocated bytes of 28% (search), 30% (db) and 38% (mtrt).

Figure 1 shows the performance speedup along with the 95% confidence intervals that we obtain through ITM. This is done as a function of the MRT and LLMRT thresholds. We observe that for some benchmarks, ITM results in a small statistically significant performance degradation: 2.5% (compress), 1.5% (javac) and 1% (SOR). This suggests that memory fragmentation due to ITM has a larger impact on overall performance than the reduction in memory footprint for these benchmarks. A number of benchmarks show a significant performance improvement: db (5%), mtrt (6%) and sparse (up to 23%). For these benchmarks, the reduction in memory consumption has a larger impact than the increased memory fragmentation. which results in a significant performance speedup. Two other benchmarks show small (but significant) performance improvements: moldyn (1.7%) and pseudojbb (1.8% for $MRT = 1\%$ and an infinite LLMRT; other MRT and LLMRT values yield statistically insignificant speedups). For all remaining benchmarks, ITM has no statistically significant impact on overall performance.

V. Prior work

A number of related research studies have been done on characterizing the memory behavior of Java applications, such as [Shuf01], [Diec99].

Venstermans *et al.* [Vens05] report that objects are nearly 40% larger in a 64-bit VM compared to a 32-bit VM, and that for non-array objects, the increased header size on 64-bit accounts for half the object size increase.

Bacon *et al.* [Baco02] present a number of header compression techniques for the Java object model on 32-bit machines. The Big Bag of Pages (BiBOP) approach was proposed by Steele [Stee97] and Hanson [Hans80]. Dybvig *et al.* [Dybv94] propose a hybrid system where some objects have a type tag in the least-significant bits and where other objects follow the BiBOP typing.

Other related research on memory reduction has been done, such as heap compression [Chen03], object compression [Chen05], [Shuf02], pointer compression [AT04], [Latt05].

References

[AT04] A. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. Lewis, B. Murphy, and J. Stichnoth. Improving 64-Bit Java IPF Performance by Compressing Heap References. In *CGO*. IEEE Computer Society, March 2004.

[Baco02] D. Bacon, S. Fink, and D. Grove. Space- and Time-Efficient Implementation of the Java Object Model. In *ECOOP*, volume 2374, 2002.

[Chen03] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *OOPSLA*, pages 282–301. ACM Press, 2003.

[Chen05] G. Chen, M. Kandemir, and M. Irwin. Exploiting frequent field values in java objects for reducing heap memory requirements. In *VEE '05*, pages 68–78, New York, NY, USA, 2005. ACM Press.

[Diec99] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *ECOOP*, 1999.

[Dybv94] R. Dybvig, D. Eby, and C. Bruggeman. Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages. Technical Report 400, Indiana University Computer Science Department, March 1994.

[Hans80] D. Hanson. A Portable Storage Management System for the Icon Programming Language. *Software—Practice and Experience*, 10(6), 1980.

[Latt05] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, New York, NY, USA, 2005. ACM Press.

[Shuf01] Y. Shuf, M. Serrano, M. Gupta, and J. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS ICMMCS*, June 2001.

[Shuf02] Y. Shuf, M. Gupta, R. Bordawekar, and J. Singh. Exploiting prolific types for memory management and optimizations. In *POPL*. ACM Press, 2002.

[Stee97] G. Steele, Jr.. Data Representation in PDP-10 MACLISP. Technical Report AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September, 1997.

[Vens05] K. Venstermans, L. Eeckhout, and K. De Bosschere. 64-bit versus 32-bit Virtual Machines for Java. *Software—Practice and Experience*, 2006. In press, http://www3.interscience.wiley.com/cgi-bin/abstract/111082816/ABSTRACT.