The second half of the chessboard
Raymond Kurzweil

Dankwoord

Vooraleer je tot het schrijven van een doctoraat komt, heb je reeds een hele levensweg afgelegd. Onderweg is het niet altijd even gemakkelijk om niet verloren te lopen: soms kom je aan een splitsing, een obstakel of een doodlopend eind. Maar steeds was er wel iemand om mij te helpen mijn weg verder te zoeken. Ik ben dan ook veel dank verschuldigd aan heel wat mensen.

In de eerste plaats aan prof. Koen De Bosschere en prof. Lieven Eeckhout, mijn promotoren, voor de mogelijkheid om mijn doctoraatsonderzoek te verrichten in hun onderzoeksgroep. Ik wens hen ook te bedanken voor de begeleiding van mijn onderzoek, voor de vele stimuli en steun en voor de goede raad en inzichten die zij mij bijbrachten.

Verder wil ik ook de andere leden binnen de Paris-onderzoeksgroep bedanken voor de goede onderzoekssfeer die er heerst. Met in het bijzonder de andere professoren, prof. Jan Van Campenhout, prof. Erik D'Hollander en prof. Dirk Stroobandt, voor het geven van constructieve opmerkingen en tips.

Mijn collega-onderzoekers Dries, Andy, Davy, Tom, Jonas, Michiel en Marc, waarmee ik ooit een bureau deelde, zou ik willen bedanken voor de aangename tijd samen, voor hun hulp en collegialiteit. Ook al mijn andere collega's zou ik graag bedanken omdat ik bij hen altijd terecht kon met vragen allerhande.

Verder zou ik ook Dr. David Bacon willen bedanken, die voor mij nieuwe onderzoekswegen opende door mij de kans de geven 2 maanden onderzoek te verrichten in het prestigieuze IBM T.J. Watson Research Center te New York.

I would also like to thank Dr. David Bacon because he opened up new research opportunities for me by giving me the opportunity to do a 2 month internship at the prestigious IBM T.J. Watson Research Center in New York.

Natuurlijk wil ik de leden van mijn examencommissie niet verge-

ten, die ik wil bedanken voor de interesse in mijn werk, het grondig lezen van mijn thesis, het geven van opmerkingen en voor de beoordeling van mijn werk.

Furthermore, I would like to thank the members of my PhD commission for the interest in my work, for reading my thesis carefully, for delivering comments and for evaluating my work.

Ook aan de Universiteit Gent ben ik dank verschuldigd, omdat deze het voor mij financieel mogelijk maakte een doctoraatsmandaat van 4 jaar op te nemen via het Bijzonder OnderzoeksFonds (BOF).

Graag zou ik ook nog een heleboel andere mensen willen bedanken die mij ook buiten mijn doctoraatsonderzoek gesteund hebben en/of mij nauw aan het hart liggen, zoals mijn ouders, mijn zus, mijn vriendin Ilse en alle andere vrienden. Met in het heel bijzonder mijn ouders, die mij al heel mijn leven lang gesteund hebben en mij zoveel gegeven hebben. Bedankt om dit alles mogelijk te maken.

Kris Venstermans
Gent, 29 mei 2007

Examencommissie

Prof. Ronny Verhoeven, voorzitter
Onderwijsdirecteur Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Jan Van Campenhout, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Koen De Bosschere, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen
Universiteit Gent

Prof. Theo D'Hondt
Vakgroep Computerwetenschappen, Faculteit Wetenschappen
Vrije Universiteit Brussel

Prof. Matthew Hertz
Department of Computer Science
Canisius College, Buffalo, NY, USA

Dr. Bilha Mendelson
Code Optimization Technologies Department
IBM Research Lab, Haifa, Israel

Samenvatting

Computerprogramma's worden complexer telkens als er nieuwe, krachtigere computerchips op de markt komen. Bij elke nieuwe generatie computerchips tasten programma-ontwikkelaars altijd opnieuw de grenzen van die generatie af. Tegelijk met het complexer worden van computerprogramma's, vereisen deze programma's ook steeds meer computergeheugen. Door die stijgende geheugenvraag werd het aantal adresseringsbits in de loop der jaren aangepast. Hedendaagse desktop systemen op de consumentenmarkt zijn allemaal uitgerust met een 64-bit CPU (Central Processing Unit), terwijl enkele jaren geleden dergelijke systemen nog bijna uitsluitend van 32-bit processoren waren voorzien. De 64-bit adresseerruimte is gigantisch groot in vergelijking met de 32-bit adresseerruimte.

Groter betekent echter niet steeds beter: als we 32-bit en 64-bit computersystemen vergelijken, merken we dat beide zowel voor- als nadelen hebben. Het meest effectieve prestatievoordeel van 64-bit systemen ligt in het aanwezig zijn van extra 64-bit instructies in de ISA (Instruction Set Architecture). Het belangrijkste nadeel van 64-bit computersystemen is dat zij meer geheugen gebruiken omdat verwijzingen (pointers) tussen verschillende objecten nu met 64 bits voorgesteld worden. De extra 64-bit instructies geven enkel een prestatievoordeel voor programma's die bewerkingen doen op grote gehele getallen, terwijl het nadeel van het grotere geheugengebruik bijna steeds zal optreden omdat objectverwijzingen heel frequent gebruikt worden in moderne programmeertalen.

In deze thesis zullen we twee technieken introduceren die het geheugengebruik van 64-bit programma's zullen verbeteren in het kader van een Java Virtuele Machine (JVM). We kiezen een omgeving die objectgeoriënteerd is, omdat objectgeoriënteerd programmeren heden ten dage een heel populair programmeerparadigma is, en omdat er veel objectverwijzingen voorkomen in een dergelijke omgeving. Vooraleer

we de nieuwe technieken introduceren, zullen we eerst een karakterisering maken van de impact van de transitie van 32-bit naar 64-bit systemen op het geheugengebruik en op de prestatie. Deze karakterisering omvat o.a. het opmeten en vergelijken van de objectgrootte van zowel array als niet-array objecten voor 32-bit en 64-bit systemen. We stellen vast dat de gemiddelde objectgrootte met 45.3% toeneemt in een 64-bit systeem ten opzichte van een 32-bit systeem. We zullen de belangrijkste redenen voor deze toename identificeren en bestuderen voor verschillende geheugenbeheersystemen. We bestuderen verder ook het groeien en inkrimpen van de heap, en we bemeten het verschil in aantal geheugensaneringen tussen 32-bit en 64-bit systemen voor een vaste grootte van de heap. Uit deze karakterisering leiden we af dat objectverwijzingen zowel in de objectdata als in de objecthoofding bestaan, en dat ze beide verantwoordelijk zijn voor een groot deel van de totale geheugenomvang van objecten in 64-bit systemen.

De eerste techniek die we voorstellen om het geheugengebruik van 64-bit systemen te reduceren, spitst zich toe op de objectdata. We comprimeren objectverwijzingen die voorkomen in de datavelden. Om deze objectverwijzingen te kunnen comprimeren, zullen we een aantal zaken moeten onderzoeken: (i) decomprimeren moet steeds correct kunnen gebeuren en dus zal er een opvangnet moeten voorzien worden voor in het geval een objectverwijzing niet comprimeerbaar is; (ii) speciale waarden voor objectverwijzingen, zoals de `null` verwijzing, moeten gedefinieerd worden in gecomprimeerde vorm; (iii) als het geheugenbeheersysteem objecten verplaatst, dan moet het mechanisme dat de objectverwijzingen tijdens het verplaatsen up-to-date brengt, aangepast worden om met gecomprimeerde objectverwijzingen te kunnen functioneren. Verder zullen we ook steeds rekening houden met de efficiëntie bij het implementeren van bovenstaande bemerkingen en stellen we optimalisaties voor om de overhead van onze techniek verder te beperken.

De tweede techniek die we voorstellen ter reductie van het geheugengebruik van 64-bit systemen, focust op de objecthoofding. We onderzoeken welke componenten er zich allemaal bevinden in de objecthoofding en, op basis van hun functie of gebruik, zullen we voor elke component een alternatieve voorstellingswijze introduceren die ons toelaat om die specifieke component uit de objecthoofding te verwijderen. Op deze manier verminderen we de grootte van de objecthoofding initieel van 16 naar 4 bytes en vervolgens elimineren we ook de laatste 4 bytes. Om deze geheugenreducerende techniek toe te passen, hebben

we in eerste instantie profielinformatie nodig. Deze profielinformatie wordt verzameld tijdens een offline uitvoering van het programma. Doordat offline technieken niet altijd handig zijn om te gebruiken in een virtuele uitvoeringsomgeving, stellen we ook een volledige online variant van de techniek voor.

Ter conclusie kunnen we stellen dat we in deze thesis het geheugengedrag van 64-bit Java Virtuele Machines bestuderen en optimaliseren. De belangrijkste bijdragen van dit werk kunnen als volgt samengevat worden: een gedetailleerde karakterisering van het geheugengebruik en van de prestaties van 32-bit en 64-bit Java programma's; twee technieken die het geheugengebruik van 64-bit programma's reduceren: één techniek die zich toespitst op de datavelden in objecten en een andere techniek die zich toespitst op de objecthoofding.

Summary

Modern computer programs increase in complexity each time more powerful computer chips are constructed. And with each generation faster and more efficient computer chips, program developers design applications that search the limits of that system. As computer programs get more and more complex, they also tend to demand more memory. Due to the increasing memory demands, the number of address bits were increased over the years. Currently most consumer market desktop systems are equipped with 64-bit CPUs, while a few years ago most such systems still had a 32-bit microprocessor. The 64-bit address space is enormous compared to the 32-bit address space.

But bigger is not always better: if we compare 32-bit with 64-bit computer systems, both have their advantages and disadvantages. The most prominent advantage of 64-bit computer systems in terms of performance is the availability of extra 64-bit integer instructions. The most important disadvantage is that applications tend to use more memory because of the 64-bit representation of pointers. While the extra integer instructions will only give benefit to applications that perform computation on such large integers, the increased memory usage disadvantage will certainly affect most programs since pointers are very heavily used in modern programming languages.

In this dissertation we will propose two techniques that improve the memory usage of 64-bit applications in the context of a Java Virtual Machine. We choose an Object Oriented environment because Object Oriented Programming is a very popular program paradigm these days and because many pointers exist in such an environment. Before we start optimizing the memory behavior of 64-bit applications, we first characterize the memory usage and overall performance impact of the transition from 32-bit to 64-bit computing for Java applications. Our characterization of the memory usage includes a measurement and comparison of the average object sizes of array and non-array objects

for 64-bit and 32-bit computing. We observe that the average object size increases by 45.3% in 64-bit mode compared to 32-bit mode. We identify the main causes for this size increase and study them for different memory managers. We also study the heap growth and measure the difference in the number of garbage collections between 64-bit and 32-bit computing systems for a fixed heap size. From our characterization, we observe that pointers exist in the object data as well as in the object header, and that both are responsible for a major part of the object's size in 64-bit systems.

The first memory reduction technique that we propose, focuses on the object data. It compresses pointers inside object data fields. In order to compress pointers, a number of issues needs to be investigated: (i) decompression needs to be correct at all times, so some kind of saveguard needs to be implemented in case a pointer can not be compressed. (ii) a good compressed representation of special pointer values, like the `null` pointer, has to be constructed and (iii) if the memory manager moves objects, its pointer update mechanism needs to be adapted so that it can handle compressed pointers. When solving these above issues, we will also take efficiency into account and propose optimizations to reduce the overhead of the compression technique.

Our second memory reduction technique concentrates on the object header. We investigate all the object header components and based on their usage/functionality, we propose for each a different alternative representation that allows us to remove that component from the object header. This way we first reduce the size of the object header from 16 to 4 bytes and next we also remove the remaining 4 bytes. Initially this memory reduction technique requires an offline profiling run to collect information about the allocation behavior of the application. However, since offline techniques are sometimes less appealing for a Virtual Execution Environment, we will also propose a variant that is entirely online.

In conclusion, this research studied the memory management of 64-bit Java virtual machines. The main contributions described in this dissertation can be summarized as follows: a detailed memory usage and overall performance characterization study of 32-bit and 64-bit Java workloads; two techniques for reducing the memory usage of 64-bit applications: one technique that focuses on the object data fields and one technique that focuses on the object header.

Contents

List of Tables

List of Figures

List of Abbreviations

BiBOP	Big Bag Of Pages
D-TLB	Data Translation Lookaside Buffer
DCX	Data Compression eXtensions
EB	Exa-Byte
EM64T	Extended Memory 64 Technology
FCP	Favorable Collection Point
GB	Giga-Byte
GC	Garbage Collection
ISA	Instruction Set Architecture
JDK	Java Development Kit
JGF	Java Grande Forum
JIT	Just-In-Time
JVM	Java Virtual Machine
KB	Kilo-Byte
LAT	Long Address Table
LLMRT	Long-Lived Memory Threshold
LOS	Large Object Space
LSB	Least Significant Bit
MB	Mega-Byte
MRT	Memory Reduction Threshold
OO	Object Oriented
ORA	Object-Relative Addressing
OS	Operating System
RVM	Research Virtual Machine
STVA	Selective Typed Virtual Addressing
TIB	Type Information Block
TLB	Translation Lookaside Buffer
TVA	Typed Virtual Addressing
VEE	Virtual Execution Environment
VM	Virtual Machine

Chapter 1

Introduction

The second half of the chessboard
Raymond Kurzweil

The phrase "The second half of the chessboard"is derived from the fable of an ancient Chinese mathematician, who did a great deed for the emperor of China. As a reward for the great deed, the emperor promised the mathematician anything in his empire he could wish for. The mathematician replied he'd wish every day an amount of rice being placed on a square of his chessboard. The first square should contain one grain, and each successive square would have to contain the double amount of the prior, and so until all 64 squares were filled. The Emperor quickly granted this seemingly humble request.

After the first half of the chessboard was filled, the total number of grains of rice became (1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 + 1024 + ... + 2.147.483.648), or $2^{32} - 1$. *This amount corresponds to about one field of rice and is considered economically insignificant to the emperor of China. It was as they progressed through the second half of the chessboard that the situation quickly deteriorated. The second half should contain* $(4.294.967.296 + ... + 9.223.372.036.854.775.808) = (2^{64} - 2^{32})$. *That is more rice than could ever be grown on the planet in the lifetime of the emperor. One version of the story claims the emperor going bankrupt, another version says the mathematician lost his head.*

1.1 Towards 64-bit computing

When people use the term 64-bit computing, it's not always clear what they mean. Most often, the width of the instructions, registers and/or addresses is meant. For the purposes of this text, it means executing applications on a processor with 64-bit registers and 64-bit virtual addresses.

Although we are currently in the transition from 32-bit to 64-bit computing in general-purpose systems, 64-bit technology is not new. A variety of high-end servers and applications like supercomputing and database management systems, have been using 64-bit hardware for over fifteen years. Companies that need to process huge amounts of data use 64-bit servers, because these servers can support both a large amount of data as well as large files.

Today, the market for consumer applications is making the transition towards 64-bit computing. The reason why it was running behind, is that this market was dominated by IA-32 instruction set architecture (ISA) based hardware. Motorola produced a 64-bit microprocessor in 1998: the PowerPC 620 processor. However, it was not until AMD extended the IA-32 ISA to the x86-64 ISA in its Athlon64 [32] and Opteron [47] microprocessors, that the market for consumer applications has been quickly catching up. Intel also followed with 64-bit x86 processors; they call their extension of the IA-32 ISA the Intel Extended Memory 64 Technology (EM64T) [35].

This currently widespread use of 64-bit technology is such a hype at the moment that computer system vendors have no trouble convincing consumers that the advantages of 64-bit computer systems are worth switching from 32-bit to 64-bit technology. There are still many 32-bit applications, but thanks to the backward compatibility of most 64-bit systems, these 32-bit applications can run unmodified on these new 64-bit systems.

Unfortunately, there are a few drawbacks to 64-bit systems supporting backward compatibility. First, there is the inconvenience for users that applications compiled in 32-bit mode and applications compiled in 64-bit mode can not be linked together. Second, since mixed-mode applications can not be linked together, each library needs to be kept twice. And third, special precautions need to be taken to support both 32-bit and 64-bit modes together: either a hardware compatibility mode—the processors support an older 32-bit ISA as well as the new

64-bit ISA—(e.g., AMD64 [1], EM64T [31], PowerPC64 [58]) or software emulation (e.g., IA-32 Execution Layer [10]), or an actual implementation of a 32-bit processor core within the 64-bit processor die (e.g., Intel Itanium [2]). The actual implementation of a 32-bit core occupies precious space on the processor die, while software emulation may face the disadvantage of being slower, due to the extra software layer. While it is not a prerequisite for the latter two cases, the support for a compatibility mode does require the 64-bit ISA to be a superset of the 32-bit ISA.

1.2 The pros and cons of 64-bit computing

We now discuss the advantages and disadvantages of 64-bit computing. We identify two advantages and two disadvantages that have a potentially serious impact on performance.

1.2.1 The pros of 64-bit computing

Obviously, the major advantage of 64-bit computing is its very **large address space**. A 32-bit machine is limited by a 4 GB address space, this is because the machine's word size is 32 bit and addresses are typically stored as one machine word. By switching to 64-bit machines, the machine word doubles in size, while the associated address space gets squared, i.e., a 64-bit machine can theoretically address 16 EB (exabyte). This is astronomically large compared to the current usage of memory. If we presume that current systems are equipped with 16 GB, and that we would double the amount of memory usage every two years, even then it would take about 60 years before we hit the new limit.

The second advantage is the availability of 64-bit wide registers and the **extension of the Instruction Set Architecture** (ISA) with extra 64-bit integer instructions. Applications that use 64-bit integer numbers can benefit a lot. A large 64-bit integer value now fits in one general purpose register, relinquishing the need for extra instructions to process intermediate results. With fewer instructions executed, execution time is expected to improve. Table 1.1 illustrates this for the multiplication of two 64-bit integer values. The last two columns show the number of instructions needed on a 32-bit processor and a 64-bit processor, respectively. This table shows that a 32-bit machine requires 6 times

Table 1.1: Resources required to first load two 64-bit integers, multiply them, and finally store the lowest 64-bit integer result.

operation	on 32-bit processor	on 64-bit processor
Load two 64-bit integers	4 32-bit registers 4 load instructions	2 64-bit registers 2 load instructions
Multiply two 64-bit integers	4 multiply instructions 2 addition instructions	1 multiply instruction
Store one 64-bit result integer	2 32-bit registers 2 store instructions	1 64-bit register 1 store instruction

more instructions than a 64-bit machine to perform a multiplication of two 64-bit integers.

We would like to note that the transition to 64-bit computing triggered some ISAs to be extended even further. Since 64-bit support requires new compilers and new operating systems, it was a great opportunity to make changes that would otherwise also require new tool chains. For example in the x86-64 ISA, 8 extra general purpose registers were added in 64-bit mode (the number of registers is doubled). This of course has a serious impact on performance, since registers are the fastest form of memory available in current processors. As reported in [44], applications execute about 25% faster on average when running in 64-bit mode on the AMD Athlon 64 3800+. However, since these extra registers are not part of the 32-bit to 64-bit address space transition, we believe that such comparison is not entirely fair if one is only interested in the impact of the transition of a 32-bit to a 64-bit address space. We prefer not to take these additional 64-bit ISA extensions into account. This thesis will focus on the impact of 64-bit addresses on performance, rather than the additional ISA extensions.

1.2.2 The cons of 64-bit computing

On the negative side, the most important disadvantage of 64-bit computing is the **increased memory footprint**. By doubling the machine word, every address requires twice the number of bits in memory. Table 1.2 shows the number of dynamically allocated data structures and their size increase when using 64-bit mode instead of 32-bit mode for 15 C benchmarks. The benchmarks are compiled using the gcc 3.3.2 compiler on an IBM Power 4 machine. All data types, except point-

Table 1.2: Dynamic memory increase between 32-bit and 64-bit C programs.

suite	benchmark	#structs	increase (%)
Grande[1]	heapsort	1	0.00
	lufact	2,004	0.03
	moldyn	3	0.00
	sor	2,001	0.07
	sparse	5	0.00
PtrDist[2]	ptrdist	32,880	4.74
	anagram	315	2.95
	ft	6,628	99.89
	ks	661	98.68
Richards[3]	bench100	14	48.15
BioPerf[4]	ce	19,011	1.51
	glimmer	135	59.74
	clustalw	4	60.00
	phylip	5	60.00
	predator	64,177	47.48
AVG		8,523	32.22

ers, have the same size in 32-bit mode as in 64-bit mode. We observe a wide variety in the number of allocated data structures across the benchmarks. A low number means fewer dynamic structures are created, and this often corresponds to a small increase in memory usage, indicating that the program is doing merely arithmetic computations on a few data structures. For example, the *Grande* applications perform arithmetic operations on large matrices. Large memory increase numbers indicate that there are a lot of pointers between dynamic structures, e.g., the benchmarks *ft* and *ks* show a close to 100% memory increase because these benchmarks almost exclusively perform operations on linked data structures.

Increased memory usage both incurs a cost component and a performance degradation component. We first discuss the cost component. One way of dealing with excessive memory usage is to provide more physical memory in the machine, however, this is costly as physical memory is a significant cost factor in today's computer systems. While in high-end servers extra memory can be installed at a greater cost,

[1] http://www.epcc.ed.ac.uk/javagrande/langcomp.html
[2] http://www.cs.wisc.edu/ austin/ptr-dist.html
[3] http://www.cl.cam.ac.uk/ mr10/
[4] http://www.bioperf.org/

there are systems that have strict space constraints, e.g., embedded systems. Designers of embedded systems are always looking for ways to shrink the hardware dimensions and to scale down the cost of their product. For these systems, increasing the amount of physical memory in the system may not be an option.

The increased memory usage may also lead to performance degradation. Although main memory access latency is typically mitigated by a hierarchy of caches, these caches are relatively small in comparison to main memory. The working set of 64-bit applications increases in size with regard to the working set of their 32-bit counterparts, without the injection of extra information (i.e., the same amount of information occupies more memory), and hence caches are more sparsely filled with useful data. This leads to more cache and translation lookaside buffer (TLB) misses, extra pressure on main memory (i.e. bandwith) and hence to overall performance degradation. This is especially of concern on heavily-loaded machines with many simultaneously running programs that are memory-intensive; overall system performance quickly deteriorates once physical memory is exhausted.

Main memory accesses tend to dominate more and more an application's execution time. This can be explained by Moore's law, which states that advances in semiconductor technology double the transistor density of integrated circuits every 18 months. Although processor performance has roughly followed this law and doubled every 18 months for the past decades, memory access time on the other hand did not follow the same speedup rate (only 7% improvement every year [37]), leading to a continuously widening of the *gap* between processor speed and memory speed. Since the *gap* is widening, the number of memory accesses — each main memory access typically takes hundreds of processor cycles — have more and more impact on the performance of the system. Next to performance degradation, the increased memory usage may potentially also increase power consumption because fewer memory banks can be shut off, there is more bus traffic, etc.

The second disadvantage of 64-bit computing is related to **address translation**. If address translation is done through the means of page tables, more levels of indirection need to be traversed in order to translate the virtual address into a physical address. In the x86 ISA family for example, a 2-level page table is used, while on the x86-64 architecture there exists a 4-level page table [1]. Because each entry in a page table is now 64 bit wide, only half of the entries can be stored in an equally

sized page. With the 4-level page scheme, only 48 bits of the virtual address space can be supported. In order to support the entire 64-bit address space, extra levels are needed. All these extra indirections of course slow down the retrieval of a page from memory and thus slows down TLB miss handling. In fact, the x86-64 Linux port already tries to limit the overhead by currently only using three of the four levels, providing only a 39-bit address space for user processes [46]. The PowerPC microprocessor family does not use a hierarchical page table, but instead uses a reversed page table system [56], needing a hashing scheme to find the corresponding physical address of a virtual address. The Intel Itanium processor provides both a linear and a hashed mechanism for hardware page walking [18].

1.3 Assessment: 64-bit computing or not?

We outlined the transition of 32-bit technology to 64-bit technology and the existence of different modes on the latter and we listed a number of advantages and disadvantages related to 64-bit technology. Now we will reason about these technologies and modes and make an assessment about 64-bit technology.

If an application requires a 64-bit virtual address space, it is obvious there is only one path to walk. Otherwise, if there is no such requirement, a choice can be made to run it either on a 32-bit system, or on a 64-bit computer system in 64-bit mode, or on a 64-bit computer system in 32-bit mode (if available). For the rest of this discussion we will presume the existence of a 32-bit compatibility mode on a 64-bit computer system which does not introduce an extra performance penalty compared to running on a 32-bit computer system[5].

We will now discuss the choice between the three options just presented, using Table 1.3. This table gives an overview of the advantages and disadvantages of 64-bit computing, applied to each choice. First, the choice between a 32-bit computer system versus a 64-bit computer system in 32-bit compatibility mode will be elucidated, followed by a discussion of 32-bit mode versus 64-bit mode on a 64-bit computer system. We will argue why 64-bit computing is not always better, explain

[5]We will not take into account performance penalties caused by running 32-bit mode on 64-bit systems. Some 64-bit architectures such as the AMD64 [25] do not incur additional overhead when running 32-bit programs in 32-bit compatibility compared to an equivalent 32-bit architecture.

Table 1.3: Overview advantages/disadvantages of 32-bit/64-bit computing.

	32-bit hardware	64-bit hardware	
	32-bit mode	compatibility mode	64-bit mode
Address Space	-	+	+
Extended ISA	-	-	+
Memory footprint	+	+	-
Address translation	+	-	-

the main cause of why this is the case, and throw light on the circumstances in which this cause is most likely problematic.

1.3.1 32-bit computing systems versus 32-bit compatibility mode on 64-bit computing systems

At first it might look as if there is no difference at all between a 32-bit computing system and a(n) (equivalent) 64-bit computing system in compatibility mode, because we stated earlier that no extra performance overhead is incurred to run 32-bit compatibility mode on 64-bit computer systems. But some subtle differences do exist. We will now discuss each of the advantages/disadvantages of Table 1.3:

- The first advantage on the 64-bit system (Table 1.3) is related to the virtual address space. The address space on a 32-bit machine is limited to less than 4 GB in practice, since a large part of the 4 GB address space is already reserved by the operating system (OS) for kernel code and shared libraries, leaving only 2 to 3 GB for user purposes. Although in compatibility mode, the virtual address space of a 64-bit machine is limited to 4 GB, the application can now use the entire 4 GB because compatibility mode on a 64-bit machine is running a 64-bit OS. The OS sees the entire 64-bit address space, and can map the 4 GB virtual address space of the 32-bit application to a different 4 GB segment from the first 4 GB segment which is typically partially used by the OS.

- The second item of Table 1.3, the extended ISA on 64-bit systems, does not apply in compatibility mode nor on 32-bit systems, since these extensions are simply not available.

- A 32-bit application run in compatibility mode will not suffer

from the third item either, an increased memory footprint, because it was compiled as a 32-bit application, and, by consequence, it will use the same amount of memory as running it on a 32-bit system.

- The fourth item related to address translation might apply since the 64-bit system is running a 64-bit OS that has access to the entire virtual address space. However, since there are systems that use hashing schemes (e.g., PowerPC, Itanium), which are similar for both systems, this disadvantage can be bypassed.

In conclusion, there are only subtle differences between a 32-bit computer system and a 64-bit computer system running in 32-bit compatibility mode. However, there is some slight advantage to the 64-bit machine running in compatibility mode.

1.3.2 32-bit compatibility mode versus 64-bit mode

Now we compare the 32-bit compatibility mode with the 64-bit mode on a 64-bit computer system. We will do this once again by discussing each of the four advantages/disadvantages listed in Table 1.3:

- Because we are discussing applications that have no need for a 64-bit address space, the first item of Table 1.3 does not apply. Both modes provide at least 4 GB virtual address space to the application.
- The second item about the extended ISA, on the contrary, is only available in 64-bit mode, so applications can only benefit when compiled for 64-bit mode.
- The third item of Table 1.3, concerning the memory footprint, applies also. There is a clear disadvantage for 64-bit mode, having an increased memory usage.
- The last item about address translation does also not apply in this discussion, since both modes run on the same machine with the same 64-bit OS.

To conclude, we can say that for applications requiring less than 4 GB virtual address space, running in 32-bit mode will benefit from

a smaller memory footprint, while running in 64-bit mode might improve performance due to the extended ISA. As there are always pointers in real-life applications, the overhead of the increased memory usage will always occur, while the extended ISA advantage will only be exploitable for applications using large integers.

1.3.3 64-bit mode: the all-around solution?

Although 64-bit computing is the future trend, the previous discussion suggested that many applications that do not require more than 4 GB of memory, will still perform best in 32-bit mode. This is a little inconvenient as two modes might need to be supported for a long time to come. We already summarized some drawbacks of supporting two modes in section 1.1, namely that object files compiled in different modes can not be linked together, and that therefore two copies of each programming library are needed. It would be better to have to support only one mode. One mode would simplify the computer hardware and the operating system. Having only the 64-bit mode would however lead to a substantial increase in memory usage, as discussed above. This increased memory usage is the obstacle that keeps us from switching *en masse* to 64-bit computing.

Implications on Object-Oriented Languages. The inflated pointer size is the major cause of increased memory usage when transferring from 32-bit computing to 64-bit computing. The *pointer* concept did not exist in the early fortran77 language, but is widely used in C programs to access and manipulate data structures. Object-Oriented (OO) programming languages go one step further and add extra functionality to the data structures, which are now called *objects*. Each object may receive messages, may process data, and may send messages to other objects. Objects often contain pointers to other objects, and, as such, when messages are passed between objects, chains of pointers are chased in the process.

If we go yet one step further in the chain of software development, we see that in modern software development technology next to the language, an environment gets created in which an application resides. A Virtual Execution Environment (VEE) shields off the application from the underlying platform, by presenting a virtual machine (VM) that provides platform independence to the application. VEEs are very

Table 1.4: Dynamic memory increase between 32-bit and 64-bit Java programs.

suite	benchmark	#structs	increase (%)
Grande[6]	heapsort	82	0.00
	lufact	2,086	0.03
	moldyn	9,063	15.15
	sor	2,084	0.03
	sparse	90	0.00
	search	34,517,857	27.18
SPECjvm98[7]	jess	32,178,443	71.74
	db	13,267,280	91.19
	javac	23,501,089	60.01
	jack	35,490,673	55.26
DaCapo[8]	antlr	12,966,988	29.17
	fop	1,180,024	47.20
	hsqldb	11,813,470	87.26
	pmd	154,355,087	78.88
SPECjbb2000[9]	pseudojbb	74,902,214	49.20
AVG		26,279,102	40.82

popular nowadays, see for example the Java Virtual Machine and the .NET environment. Virtual execution environments provide benefits in terms of portability, improved security and better resource management. For example, the Java Virtual Machine (JVM) manages its own object allocation and deallocation: reclamation of dead objects from the heap is done by a garbage collector. In order to support garbage collection and some other object functionality like virtual method dispatching, locking and hashing, the JVM needs to be able to keep track of object-specific information. This information is typically stored in a header attached to each object.

With regard to memory management, OO-languages tend to use more dynamic memory than non-object-oriented languages like C. Calder et al. [17] compare C to C++ applications and observed that C++ applications allocate 4x more dynamic memory than C applications, on average. A first reason is obviously the extra object headers

[6] http://www.javagrande.org
[7] http://www.spec.org/jvm98
[8] http://www.spec.org/jbb2000
[9] http://dacapobench.org/

which require memory chunks to be larger, and second, since everything is an object in an OO-language like Java, more chunks of memory are likely to be allocated [33]. Table 1.4 shows the number of objects allocated for 15 Java benchmarks and the increase in dynamic memory utilization when making the transition from 32-bit to 64-bit pointer representation. Comparing Table 1.4 to Table 1.2 clearly shows that Java applications typically allocate more dynamic data structures. The added object headers typically contain pointers, and hence make the total memory increase in an OO-language like Java even more drastic (e.g., Table 1.4 shows a 40.82% increase on average, compared to a 32.22% increase in Table 1.2). The low increase in dynamic memory utilization for most of the *Grande* applications can be explained by the nature of these applications—they operate on large arrays and do not manipulate pointers frequently.

1.3.4 Conclusions

As a general conclusion of whether it is worth adapting to 64-bit computing or not, we can make a few statements.

First, **64-bit technology cannot be stopped.** All major manufacturers on the general-purpose consumer market have integrated 64-bit technology in all of their products. As 64-bit computer systems with a 32-bit compatibility mode seem to have a slight advantage over pure 32-bit computer systems, the current hype on the consumer market might make sense. Consumers who buy 64-bit technology do not have to give up performance for applications that can run in 32-bit mode. In addition, having a 64-bit computer makes them ready for future applications which might need the extended virtual address space. Moreover, manufacturers can produce only one architecture (that supports both 32-bit and 64-bit applications) instead of two. Due to the economy of scale, production costs decrease, which is again favorable for the consumers.

Second, **64-bit computing is not always better than 32-bit computing.** The statement that consumers would not have to give up performance on 64-bit computer systems is only valid assuming compatibility mode. In 64-bit mode, the real benefit depends on the application. If the application cannot benefit from the address space increase or the 64-bit operations, it will probably suffer from the increased memory footprint.

Third, **OO-languages and VEEs suffer more from the disadvantages of 64-bit computing.** If 64-bit mode is used, the increased memory size is the largest cost. The memory increase effect is the largest for applications with many strongly connected objects, but since this effect is even intensified by OO-languages and execution environments with memory management, 64-bit computing is expected to be worse in such an environment.

In general, without space saving techniques specifically targeted at 64-bit platforms, good judgement is in order whether to compile an application in 32-bit mode or in 64-bit mode. If each bit of the 64 bit of an address corresponds to a square of the mathematician's chessboard in the Chinese fable, one should be careful not to make the same mistake as the emperor, entering the second half of the chessboard unadvised.

1.4 Goal and contributions of this thesis

Ideally, applications should only take advantage of 64-bit computing without its associated cost. In reality though, these two factors are connected, and a trade-off has to be made between extra addressability and extra memory usage. In this dissertation we make such a trade-off and search for a way in between: use the advantages of 64-bit computing while avoiding paying the penalties, or in other words, provide a 64-bit address space for applications that need it, while minimizing excessive memory usage because of 64-bit pointers for applications that do not require a 64-bit address space. This thesis makes three major contributions:

- **A detailed 64-bit memory usage characterization study.** Our first contribution [69] is to characterize the impact of the transition from 32-bit to 64-bit computing on memory usage and overall performance for Java workloads. We found that objects grow on average with about 45.3% when going from a 32-bit to a 64-bit Java Virtual Machine. We identified the inflated pointer size of 64-bit pointers as the major cause of this increased object size. From this characterization, we observe that pointers exist in the object data as well as in the object header and that both are responsible for a major part of this increased object size. We also study the memory usage of applications through time, and compare heap growth behaviors of a 32-bit VM and a 64-bit VM. We perform a

detailed performance analysis including cache and TLB miss rate characterization.

- **Object-Relative Addressing.** The cost of 64-bit computing, i.e., the increased memory usage, can be reduced by space saving techniques specifically targeted at 64-bit platforms. In order to achieve a smaller memory footprint, we reduce the size of each/most of the dynamically allocated memory chunks. In Java, each object is dynamically allocated. An object consists of a header and a body and both can contain references. This thesis addresses the excessive memory usage in both the header and the body.

  Our first technique concentrates on the object body. References inside the body of an object are used to create links between objects. As linked objects tend to form clusters [60], it is possible to lay out objects on the heap so that many linked objects are in close proximity [39]. If objects are known to be in close proximity, the reference can be encoded in less than 64 bit. Recall that a reference is an abstraction of a pointer, and hence its physical representation can be chosen freely. We propose a technique, called Object-Relative Addressing (ORA), which stores an inter-object reference as a 32-bit offset relative to the referencing object's virtual address. ORA results in more than 10% reduction in allocated bytes for many benchmarks while enabling applications to allocate more than 4 GB, unlike prior work in this area [3]. ORA is published in [72].

- **Selective Virtual Typed Addressing.** Our third contribution is a technique that reduces the excessive memory usage in the object header. Each object typically has a header attached for the VM to perform internal functionality such as virtual method dispatching, garbage collection, hashing, locking, etc. The object header typically contains a reference that (indirectly) reveals an object's type. In contrast to the references in the body, the purpose of this reference is not to link objects, but to identify the object's run time type. A first observation is that in real-life applications there are only several thousands of types, so using 64 bit in order to distinguish object types is likely to be excessive. Second, the 64-bit virtual address space is so huge, that it is extremely unlikely that this entire space is going to be used in the next few decade(s). Hence we could give each object type its own parti-

tion of the virtual address space, i.e., we allocate all objects of a given type in a particular memory segment. Object type identification can then be done by looking at the address of the object itself, rather than loading a reference from its header. We call this technique Typed Virtual Addressing (TVA). By doing so, the type information in the header gets redundant and hence can be removed. In order not to create excessive fragmentation due to all object types being allocated in different regions in memory, we apply TVA to a selection of objects only, hence the name Selective Typed Virtual Addressing (STVA). Next to applying STVA, we also propose techniques to remove the remaining header fields. In [70] we describe how the selection criterion of STVA can be performed offline and how we reduce the object header from 16 bytes to 4 bytes. Next, [71] extends the former paper by removing the entire object header and by proposing an online TVA selection technique. This reduces memory demands by 15% on average.

1.5 Publications

This thesis has three contributions: a characterization study of the memory usage and overall performance of 32-bit and 64-bit applications, a pointer compression technique and a header removal technique. These contributions are published in the following papers.

- The characterization study is published in the paper:

  [69] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. **64-bit versus 32-bit Virtual Machines for Java**. In Software-Practice and Experience (SPE), 36(1): pages 1-26, Jan 2006.

- The ORA pointer compression technique is further detailed in:

  [72] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. **Object-Relative Addressing: Towards Compressed Pointers in 64-bit Java Virtual Machines**. In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07), to be published. July 2007.

- The next publications describe the Typed Virtual Addressing

(TVA) and Selective Typed Virtual Addressing (STVA) techniques:

[71] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. **Java Object Header Elimination for Reduced Memory Consumption in 64-bit Virtual Machines**. In ACM Transactions on Architecture and Code Optimization, to be published, 2007.

[70] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. **Space-Efficient 64-bit Java Objects through Selective Typed Virtual Addressing**. In proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO), pages 76-86, Mar. 2006.

Besides the contributions described in this dissertation, we also performed research on garbage collection.

- **Garbage Collection Hints.** Current garbage collection (GC) techniques typically trigger when either the heap, or a part thereof, is full. In between two runs no objects get deallocated, i.e., dead objects need to wait for the next GC to be discovered and freed. The right time to collect is a balance between freeing up unused memory as soon as possible and freeing up as much memory as possible at once. GC algorithms can be made smarter when provided with liveness information from the application [41]. We present a technique, that during the execution of a program feeds back the amount of live data of a training run of that program. From this live data information, execution points can be identified, where the amount of live data reaches a local minimum. We call these execution points favorable collection points (FCPs). In particular, less dead objects occupy the heap if collections are triggered at points in time where the amount of live data is low. The proposed Garbage Collection Hints mechanism helps the VM to decide when and how to collect at these FCPs. This fourth contribution is not included in this thesis, but is published in:

  [15] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. **GCH: Hints for Triggering Garbage Collections**. In Transactions on High-Performance Embedded Architectures and Compilers, 1(1):pages 52-72, Jun. 2006.

  [14] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and

Koen De Bosschere. **Garbage Collection Hints**. In Proceedings of High-Performance Embedded Architectures and Compilers 2005 (HiPEAC'05), LNCS 3793, pages 233-348, Nov. 2005.

1.6 Overview

This dissertation is organized as follows. First, chapter 2 quantifies the increased memory usage of 64-bit computing versus 32-bit computing for Java workloads. In addition, it identifies the major causes of this increase and evaluates the impact on performance.

Since inflated references can cause a significant memory usage increase in 64-bit virtual machines, we investigate memory saving techniques in chapters 3 and 4. Chapter 3 describes the Object-Relative Addressing technique to reduce pointer sizes, while Chapter 4 describes (Selective) Typed Virtual Addressing.

Finally, Chapter 5 summarizes the main conclusions and results from this dissertation and finishes with some perspectives at future work.

Chapter 2

64-bit versus 32-bit computing: a characterization

A lie told often enough becomes the truth.
Nikolai Vladimir Ilyich Lenin

In this chapter we are interested in 32-bit versus 64-bit Java processing and its impact on performance. Many speculations have been made about the impact on performance of 64-bit versus 32-bit computing, however, few quantitative results are available. The speculations being made typically concern the impact on memory usage and the impact on execution speed. To the best of our knowledge, this thesis is the first to investigate the impact of 32-bit versus 64-bit computing in the context of Java workloads. In this chapter, we study and quantify both the increased memory requirements due to 64-bit computing and its impact on overall performance in Java.

2.1 Introduction

The introduction chapter discussed advantages and disadvantages of 64-bit computing. We concluded that 64-bit computing will become common practice, notwithstanding the fact that for some applications 64-bit computing may result in a performance penalty compared to 32-bit computing. The dominating concern when running in 64-bit mode is caused by the increased memory usage of applications. This in-

creased memory usage is even intensified if the applications are written in Object-Oriented programming languages.

The observation that 64-bit computing involves increased memory usage is not surprising. However, there is no prior work with quantitative measurements on the increased memory requirements of 64-bit versus 32-bit computing for Java applications. As such, before we tackle the increased memory usage of 64-bit Java applications in the following chapters, we feel that a detailed characterization study of the overhead of 64-bit computing over 32-bit computing is needed first. The measurements done in this chapter, describing this overhead, will form the basis that the subsequent chapters will build upon.

In this chapter, we show that the space an object occupies on the heap increases by 45.3% on average when using a 64-bit VM versus a 32-bit VM. We identify four causes for this: (i) the increased pointer size (64 bits versus 32 bits), (ii) the increased header, (iii) the increased number of bytes that need to be inserted within the objects for alignment purposes and (iv) the increased number of bytes between objects on the heap due to alignment and memory manager overhead.

We also quantify the increased number of garbage collections (GCs) performed and the increased amount of time spent during GC when run in 64-bit mode. From our experimental setup, we conclude that on average GC performs 33.4% worse while 60.1% more minor collections and 64.8% more major collections are performed in a 64-bit VM for the same heap size.

Next, we also quantify the impact on overall performance and study the behavioral characteristics of a 32-bit VM compared to a 64-bit VM using hardware performance monitors. We conclude that 64-bit computing typically results in a larger number of data cache misses at all levels in the cache hierarchy. For example, we report 21.3%, 48.8% and 66.5% increase in the number of data cache misses when comparing 64-bit computing to 32-bit computing for the L1, L2 and L3 data caches, respectively. Finally, when considering overall performance, we conclude that Java applications can run much faster in 64-bit mode if they can benefit from the extra 64-bit instructions. Most applications, however, show a slowdown of a few percentage when running in 64-bit mode compared to running in 32-bit mode.

2.2 Experimental setup

This section describes the experimental setup for this chapter and the remainder of this thesis. We discuss the virtual machine, the hardware platform and the benchmarks used.

2.2.1 Virtual machine

We use the Jikes Research Virtual Machine (Jikes RVM) version 2.3.5, extended to be able to use the entire 64-bit virtual address space. The Jikes RVM [5] is an open-source virtual machine, developed by IBM Research[1]. It runs on Linux/IA32, AIX/PowerPC, Linux/PowerPC and OS X/PowerPC. All these ports are for 32-bit machines, except for the AIX/PowerPC which was also ported to 64 bit. Jikes RVM uses a compilation-only scheme for translating Java bytecodes to native machine instructions. Upon the first invocation of a Java method, the Jikes RVM compiles the Java bytecode to native code using its baseline compiler. Whenever code is considered to be hot code, the optimizing compiler will further optimize this code using advanced compiler optimizations. One particularity of Jikes RVM is that Jikes RVM itself is written in Java, which allows for more aggressive optimizations between application code and RVM code. It also has an effect on the objects allocated. Internal objects such as those created during class loading or those created by the run time compilers are all allocated on the same Java heap. Thus unlike with conventional Java virtual machines (not written in Java) the heap contains both application data as well as VM data.

We will use four garbage collectors with Jikes RVM. We consider two basic collection strategies, namely SemiSpace which is the simplest copying garbage collector (worst heap usage) and MarkSweep which is the simplest non-copying garbage collector (worst collector-specific overhead). Furthermore, we will examine two generational collection schemes. A generational garbage collector segregates the heap in several generations. The generational garbage collectors we use, have 2 generations: a nursery generation and a mature generation. New objects are allocated in the nursery generation. At GC-time, the surviving objects from the nursery generation are copied into the mature generation (minor GC). As the mature generation fills up, less and less space becomes available for the nursery generation. If the size of the nurs-

[1] http://www.ibm.com/developerworks/oss/jikesrvm

Table 2.1: Cache hierarchy of the IBM POWER4.

cache	size	line size	associativity
L1 I-cache	64 KB	128 B	direct mapped
L1 D-cache	32 KB	128 B	2-way set assoc.
L2 unified	1.41 MB	128 B	8-way set assoc.
L3 unified	32 MB	512 B	8-way set assoc.

ery generation drops below a specified limit, a major GC is triggered and both the nursery and mature generations will be collected. The surviving objects from both generation are copied into the mature generation. Generational collectors are very popular in todays VMs, since they typically provide better performance and they decrease collector pause times [40]. The two generational collection schemes we use, are GenCopy, which is a generational version of the SemiSpace collector, and GenMS (generational MarkSweep), which is the best performing garbage collector of Jikes RVM and which in essence is a combination of SemiSpace (copying between young generation and old generation) and MarkSweep. The heap size that can be occupied before triggering a GC is controlled by two parameters. First the value of *initial heap size* will set an upperbound to the available heap size. Jikes RVM can adjust this value at run time, according to its needs and the pressure on the GC-system. However, the value will never exceed the value of the second parameter, called the *maximum heap size*.

Jikes RVM was extended to support reading the Hardware Performance Monitors per thread [66]. We make use of this extension to measure cache and TLB miss rates as well as overall performance. We measure these performance numbers for both the mutator threads and the garbage collection threads.

2.2.2 PowerPC platform

The hardware platform used in this chapter (and the entire thesis) is the IBM POWER4 microprocessor [11]. The IBM POWER4 is a 64-bit microprocessor implementing the PowerPC ISA with two cores on a single chip. Each core is an 8-issue superscalar out-of-order microprocessor capable of processing over 200 in-flight instructions at any given time. The POWER4 can be used in a multiprocessor system with

several POWER4 chips on the same motherboard. Our machine however, a 615 pSeries, only has one single POWER4 chip. The amount of RAM memory equals 1 GB. The memory subsystem of the POWER4 has three levels of cache. Each core has an L1 instruction cache (I-cache) and L1 data cache (D-cache). The L1 D-cache is a *write through* cache, which means that all data stored in the L1 D-cache is immediately stored through to the L2 cache. The L2 cache is a unified cache and is shared by the 2 cores on the chip. The L2 cache is a *write back* cache, meaning that data is not immediately written to memory—this is done upon replacement. The L3 cache is designed to be shared by multiple POWER4 chips. The L3 controller containing the tag arrays are stored on chip whereas the L3 data arrays are stored off-chip. More details on the cache hierarchy of the POWER4 can be found in Table 2.1.

The unified TLB (for both instructions and data) has 1024 entries in a 4-way set-associative structure. The effective to real address translation tables (I-ERAT and D-ERAT) operate as caches for the TLB. They are organized as 128-entry 2-way set-associative arrays. The POWER4 has standard 4 KB pages but also supports large 16 MB pages. In our measurements we only use 4 KB pages. 16 MB pages are limited in use since they are limited in number (to be provided at boot time of the machine) and are only accessible for privileged users.

In the evaluation section later in this chapter we will characterize the behavior of the 32-bit and 64-bit VM using hardware performance monitors. Current microprocessors are generally equipped with a set of specialized registers to count a variety of hardware events such as number of cycles executed, instructions executed, cache misses, branch mispredictions, etc. The AIX 5.1 operating system provides an application programming interface in the form of a kernel extension (the pmapi library[2]) to access these hardware performance counter values. This library automatically handles hardware counter overflows and kernel thread context switches. These performance counters measure both user and kernel activity.

2.2.3 Statistical analysis

Since we measure on real hardware, non-determinism in evaluation runs results in slight fluctuations in the number of execution cycles.

[2]http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftools/doc/prftools/perfmon_api.htm

Table 2.2: Benchmarks used from the SPECjvm98 suite.

db	This benchmark makes some database requests on a memory resident database.
jack	An early version of the JavaCC Java source code parser generator.
javac	The Java Development Kit (JDK) 1.0.2 Java to bytecode compiler.
jess	An expert shell system, based on NASA's CLIPS expert system, solving a set of puzzles with varying degree of difficulty.

In order to be able to draw statistically valid conclusions, we use the unpaired or noncorresponding setup for comparing means, see [53] (pages 64–69). This statistical test goes goes as follows. We first calculate the difference of the means $\bar{x} = \bar{x}_1 - \bar{x}_2$. We subsequently compute the standard deviation of the difference of the means: $s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$, with n_1 and n_2 the number of measurements and s_1 and s_2 the standard deviations in the two setups. The two setups in this chapter are the 32-bit VM and the 64-bit VM, respectively. The confidence interval can then be computed as $[\bar{x} - t_{1-\alpha/2;n_{df}}; \bar{x} + t_{1-\alpha/2;n_{df}}]$, with $t_{1-\alpha/2;n_{df}}$ the critical value of the Student t-distribution with n_{df} degrees of freedom at the $1-\alpha$ confidence level. The number of degrees of freedom is computed as follows: $n_{df} = \frac{\left(\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}\right)^2}{\frac{(s_1^2/n_1)^2}{n_1-1}+\frac{(s_2^2/n_2)^2}{n_2-1}}$.

In our experimental results, we report 95% confidence intervals. In all of our experiments, we run 15 measurements, i.e., $n_1 = n_2 = 15$.

2.2.4 Benchmarks

We selected several benchmark suites to construct our benchmark set: SPECjvm98[3], SPECjbb2000[4], Java Grande Forum (JGF)[5] and DaCapo[6].

SPECjvm98 is a client-side Java benchmark suite, for which we use the s100 input set. We were unable to run the mtrt benchmark on Jikes

[3]http://www.spec.org/jvm98
[4]http://www.spec.org/jbb2000
[5]http://www.javagrande.org
[6]http://dacapobench.org/

Table 2.3: The PseudoJBB benchmark.

pseudojbb	A variant of SPECjbb2000, which is a three-tier transaction system server benchmark, where the user interaction (first tier) is simulated by random input selection and the database (third tier) is represented by a set of binary trees. The benchmark focuses on the business logic found (middle tier). Pseudojbb runs for a fixed number of transactions (35,000) whereas SPECjbb2000 runs for a fixed amount of time. The number of warehouses goes from 1 to 8.

Table 2.4: Benchmarks used from the Java Grande Forum suite.

crypt	This benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. ($N = 50M$)
heapsort	Sorts an array of N integers using a heap sort algorithm.($N = 25M$)
lufact	Solves an N x N linear system using LU factorization followed by a triangular solve. ($N = 2,000$)
moldyn	Evaluation of an N-body model for particles interacting under a Lennard-Jones potential in a cubic space. ($N = 8,788$)
search	A program solving a connect-4 game, using an alpha-beta pruning technique. N positions are evaluated. ($N = 34,517,760$)
sor	This benchmark performs 100 iteration of successive over-relaxation on a N x N grid. ($N = 2,000$)
sparse	A N x N sparse matrix is used for 200 iterations. The sparse matrix is stored in a compressed-row format with a prescribed sparsity structure. This benchmarks exercises indirect addressing and non-regular memory references. ($N = 500K$)

RVM 2.3.5. We did not use the compress and mpegaudio benchmarks, because they allocate almost no memory. The initial heap size is set to 100 MB and the maximum heap size to 200 MB for all the benchmarks. Each benchmark is given a short description in Table 2.2.

Table 2.5: Benchmarks used from the DaCapo benchmark suite.

antlr	parses one or more grammar files and generates a parser and lexical analyzer for each
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file
hsqldb	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
pmd	analyzes a set of Java classes for a range of source code problems

Pseudojbb, see Table 2.3, is a variant of SPECjbb2000, a server-side benchmark focusing on the business logic of a three-tier system. We used increments of 1 warehouse, ranging from 1 to 8 warehouses. Instead of running for a fixed amount of time as done in standard SPECjbb2000, pseudojbb processes a fixed amount of work. We have set the transaction parameter to 35,000 units. We set the maximum heap size to 512 MB for the MarkSweep and GenMS collectors and to 768 MB for the SemiSpace and GenCopy collectors. The initial heap size is set to 256 MB.

A very different set of applications are the sequential benchmarks from the Java Grande Forum suite. A more detailed description for each benchmark can be found in Table 2.4. These so-called *Grande* applications require large amounts of memory, bandwidth and/or processing power. Examples include computational science and engineering codes, as well as business and financial models. For each of the selected benchmarks we have chosen the largest input set available. For the JGF benchmarks the initial heap size is set to 256 MB and the maximum heap size to 384 MB.

Finally, we also use the DaCapo benchmark suite [13], which exhibits more complex code, richer object behaviors and more demanding memory system requirements than the SPECjvm98 client-side benchmarks. We use the DaCapo benchmarks under version beta-2006-08. Unfortunately, we were unable to run all the DaCapo benchmarks on Jikes RVM 2.3.5; we use the 4 DaCapo benchmarks mentioned in Table 2.5. These were the only 4 DaCapo benchmarks we could run with the large input set on our version of Jikes RVM. We set the maximum heap size to 512 MB with a 100 MB initial heap size in all of the DaCapo

experiments, except for hsqldb, for which we set the maximum heap size to 768 MB for the GenCopy and SemiSpace collectors.

2.3 32-bit versus 64-bit VM

Before quantifying the performance overhead when switching from the 32-bit VM to the 64-bit VM, we will start by highlighting some key differences between the 32-bit and 64-bit versions of the Jikes RVM. We identify four key differences: (i) the extended ISA, (ii) the increased stack size, (iii) argument passing and (iv) the increased object size on the heap.

2.3.1 64-bit ISA

Basically the 64-bit PowerPC ISA is a superset of the 32-bit ISA. Only 64-bit applications can make use of the 64-bit ISA. The 64-bit ISA includes additional instructions such as arithmetic operations on 64-bit integer values, loading and storing 64-bit values to and from memory, etc. This can be beneficial for applications working on 64-bit quantities, since 32-bit compiled applications would require a sequence of instructions to compute the same result as demonstrated earlier in Table 1.1. In these cases, the 64-bit VM will need fewer native instructions. However, in other cases, the 64-bit VM might require more native instructions than the 32-bit VM. For example, manipulating 32-bit offsets in 64-bit computing needs additional operations to sign/zero extend offsets, or, loading a constant reference into a register requires extra instructions to load the upper 32 bits.

2.3.2 Increased stack size

The sizes for the different Java types when used as object fields in Jikes RVM, i.e., when allocated on the heap, are listed in Table 2.6 in the 'field size' column. This is the same in 32-bit mode as in 64-bit mode, except for the 'reference' and 'returnAddress' types which are addresses. When pushing and popping these Java types on/off the operand stack, a different number of bytes will be allocated and deallocated. As shown in the 'size on stack' column in Table 2.6, the size of the Java types doubles for all Java types, when comparing the 64-bit VM to the 32-bit VM. As such, the amount of stack space needed in

Table 2.6: Java types and their sizes measured in the number of bits when used on the heap ('field size' column) and when used on the stack ('size on stack' column).

	32-bit platform		64-bit platform	
Java types	Field size	Size on stack	Field size	Size on stack
boolean	32	32	32	64
byte	32	32	32	64
char	32	32	32	64
short	32	32	32	64
int	32	32	32	64
float	32	32	32	64
reference	32	32	64	64
returnAddress	32	32	64	64
long	64	64	64	128
double	64	64	64	128

64-bit mode is twice as much as in 32-bit mode. Note that the baseline compiler in the Jikes RVM uses the operand stack intensively for storing intermediate values—the baseline compiler nearly literally translates the Java bytecode stack processing to native stack processing. The reason that *all* types take twice as much stack space is due to the fact that all Java types use a fixed number of stack slots (requirement for the Java bytecode) and a stack slot needs to be able to host an address, whose size doubles in 64-bit mode. A detailed discussion of this issue however is out of the scope of this thesis. We refer the interested reader to [68] for more details.

2.3.3 Argument passing

Argument passing may also cause differences between 32-bit and 64-bit VMs. Passing a `long` in 32-bit mode requires 2 general purpose registers, but requires only 1 register in 64-bit mode. This reduces register pressure in 64-bit VMs.

2.3.4 Increased object size

There are four causes why objects as they are allocated on the heap are larger in a 64-bit VM than in a 32-bit VM. All four are depicted in Fig-

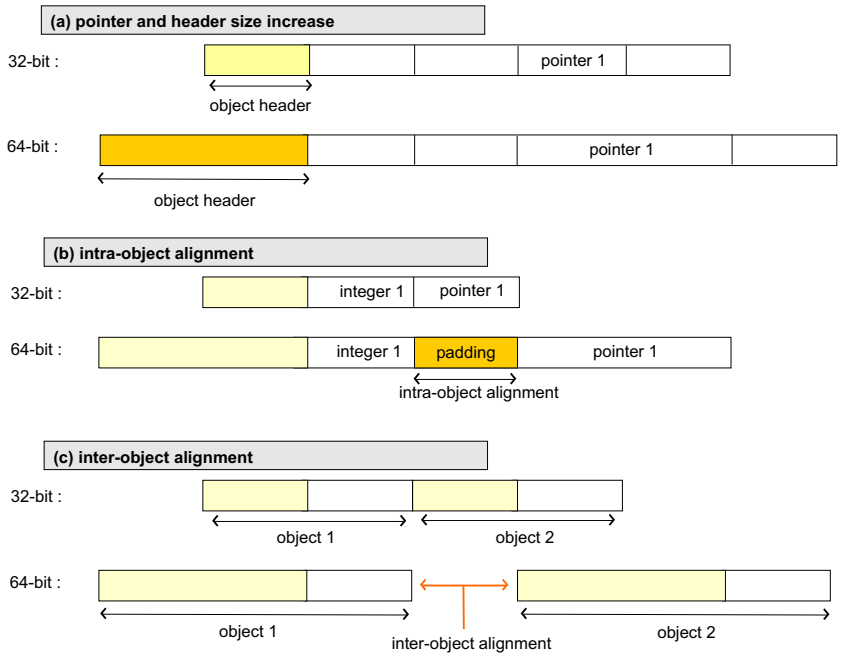

Figure 2.1: Causes for object size increase when going from the 32-bit VM to the 64-bit VM.

ure 2.1. The first reason obviously is the increased size of the pointers inside the body of an object, namely 64 bits instead of 32 bits. The second reason concerns the header. Both the first and the second reason are shown in Figure 2.1(a). The default object model in 32-bit Jikes uses 2 header words (32 bits each) and 1 extra header word for array objects containing the array's length (also 32 bits). These header words double in size in 64-bit Jikes RVM[7]. However, we observed there is room for improvement because 64-bit Jikes does not fully use all words at all times. More in particular, the array length needs only 4 bytes, 4 out of 8 bytes in the status word are only used by copying garbage collectors, and the Type Information Block (TIB) word could possibly be compressed. The third reason for the increased object size, shown in Figure 2.1(b), is the fact that additional bytes need to be allocated for alignment. On 64-bit platforms, one typically wants references to be aligned on 8-byte boundaries, whereas in 32-bit mode, alignment at 4-byte boundaries is sufficient. Note that 64-bit fields (in both 32-

[7]The first two words effectively double in size. The array length field always takes 4 bytes, but the resulting 20 bytes in 64-bit mode need 4 bytes in addition for alignment.

bit and 64-bit mode) will get aligned on 8-byte boundaries. This implies the possible existence of a hole (e.g., an unused padding field of 4 bytes) inside an object; in the remainder of this dissertation, we will refer to this padding as *intra-object alignment*. The fourth reason is the extra overhead introduced by the memory manager due to inter-object alignment—not to be confused with the intra-object alignment as just discussed. Inter-object alignment is shown in Figure 2.1(c). Inter-object alignment comes from aligning the object pointer in the allocated heap space; intra-object alignment comes from aligning the fields to the object pointer. In general, all four causes cause the object size to increase when transitioning from a 32-bit machine to a 64-bit machine. This potentially impacts cache and TLB behavior and by consequence overall performance.

2.4 Memory behavior

Now we will quantify the allocation behavior and the increase in memory usage when comparing the 64-bit VM with the 32-bit VM. We will measure the increase in overall object size.We subsequently separate our measurements for application objects and RVM objects, in order to verify whether our measurements would still hold for VMs other than Jikes RVM that are not written in Java. Next, a quantification of the memory increase caused by the memory manager is presented. Finally, we compare the heap behavior of the 32-bit VM and the 64-bit VM.

2.4.1 Average object size

As pointed out in section 2.3.4, the object size on the heap increases when comparing 64-bit and 32-bit Java computing because of four causes: (i) 64-bit versus 32-bit pointers, (ii) the header doubling in size, (iii) additional padding for intra-object alignment and (iv) extra space due to inter-object alignment.

Table 2.7 presents the average object size in 32-bit and 64-bit mode along with its relative increase. This is done for all objects (both array and non-array), array objects and non-array objects. For non-array objects, we observe an increase in size from 32-bit to 64-bit objects that is nearly constant over all benchmarks. The average increase is about 16 to 20 bytes. Recall from the previous section that for non-array objects 8

Table 2.7: Average object size (in bytes) in 32-bit and 64-bit VM mode for all objects, array objects and non-array objects.

	overall			array objects			non-array objects		
Benchmark	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)
db	25.6	48.5	89.6	136.5	248.6	82.1	17.6	34.1	94.0
jack	37.0	58.6	58.4	50.8	75.2	48.0	25.7	44.3	72.3
javac	32.9	53.9	63.8	51.7	76.2	47.4	25.6	45.3	76.8
jess	34.7	59.8	72.4	49.0	96.9	97.9	28.0	42.4	51.6
antlr	48.9	67.2	37.3	66.4	83.2	25.4	25.6	45.0	75.6
fop	64.7	87.9	35.8	124.5	153.9	23.6	28.1	46.7	66.2
hsqldb	37.8	64.3	70.0	64.4	109.8	70.6	29.2	49.3	68.6
pmd	24.5	43.7	78.1	52.8	96.8	83.2	19.7	34.6	75.8
crypt	719.7	876.5	21.8	4,425.0	4,693.5	6.1	30.8	54.3	76.1
heapsort	373.8	403.4	7.9	1,870.4	1,935.2	3.5	27.4	48.3	76.6
lufact	174.3	205.8	18.0	786.6	863.2	9.7	28.2	49.4	75.5
moldyn	44.3	72.8	64.4	93.7	152.1	62.3	29.7	49.4	66.3
search	43.5	56.1	29.0	44.3	56.6	27.8	24.8	44.7	80.0
sor	292.3	329.6	12.7	1,167.0	1,247.7	6.9	30.0	51.5	71.3
sparse	310.5	341.3	9.9	1,551.7	1,608.9	3.7	29.8	51.2	72.0
pseudojbb	33.9	52.8	55.8	44.6	63.6	42.6	27.6	46.3	67.9
Average			45.3			40.1			72.9

of these bytes come from the increased header. The remaining increase thus comes from alignment and larger pointers in the object fields.

The average relative increase in size for non-array objects is 72.9%. This is fairly high and will lead to poorer cache utilization, as will be shown in section 2.5. For array objects, the picture is different: the array object size increases by 40.1% on average. Some benchmarks have small object size increases (e.g., 3.5% for heapsort) whereas others suffer from large object size increases (e.g., 97.9% for jess). This suggests that most arrays in jess contain references (which all double in size) whereas for heapsort, most arrays do not contain references (those array objects only have a header increase). This big difference between array and non-array objects can be explained by the fact that most non-array objects are small [26]. For small objects, even a few extra bytes can lead to a severe size increase.

When considering both array and non-array objects, we observe an average object size increase of 45.3%. The average increase is the largest for SPECjvm98 suite (71.1%) and the least for Java Grande Forum suite (23.4%). DaCapo has an increase of 55.3%, sligthly above the average.

Until now, we considered all objects, both small and large objects. In the following set of measurements we focus on small objects. The reason for doing this is that in Java most objects are typically small [23] and that small objects and their placement can be manipulated easily by the memory allocator and the garbage collector. Previous work has shown that data layout is an important issue (especially for small objects) for exploiting spatial and temporal locality—for example, collocating objects that are related to each other on the same cache line can improve locality [24, 67]. The increase in small object sizes can thus affect the performance of such optimizations.

Distinguishing between small and large objects can be easily done in Jikes RVM since it maintains a Large Object Space and an Immortal Space next to (a) collector-specific space(s) (three spaces for the GenCopy collector, two spaces for the SemiSpace and GenMS collectors and one space for the MarkSweep collector). The Large Object Space, as its name suggests, is used for allocating objects that are larger than a given threshold which is 8 KB for Jikes RVM 2.3.5. A copying garbage collector will not move objects in this space. The Immortal Space contains objects that are never collected, i.e., specific RVM objects. Table 2.8 shows the average object size in 32-bit and 64-bit mode solely for objects allocated in the collector-specific space(s), i.e., objects in the Large Object

Table 2.8: Average object size (in bytes) in the collector-specific spaces in 32-bit and 64-bit VM mode for all objects, array objects and non-array objects.

	overall			array objects			non-array objects		
Benchmark	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)
db	19.2	36.1	88.1	41.8	64.2	53.8	17.6	34.1	94.0
jack	36.7	58.3	58.5	50.3	74.6	48.3	25.7	44.3	72.3
javac	32.8	53.4	62.8	51.2	74.3	45.2	25.6	45.3	76.8
jess	34.6	59.5	72.2	48.7	96.3	97.6	28.0	42.4	51.6
antlr	29.4	47.1	60.1	32.2	48.6	50.8	25.6	45.0	75.6
fop	48.1	68.5	42.5	80.9	103.6	28.1	28.1	46.7	66.3
hsqldb	32.8	54.2	65.5	43.7	69.3	58.5	29.2	49.3	68.6
pmd	21.3	37.1	74.0	30.9	51.8	67.6	19.7	34.6	75.8
crypt	36.8	61.9	68.2	69.0	97.3	41.1	30.8	54.2	76.1
heapsort	32.7	55.6	69.8	55.9	86.9	55.4	27.4	48.3	76.6
lufact	34.6	57.5	66.1	63.0	93.2	47.9	28.2	49.4	75.5
moldyn	37.4	58.9	57.3	63.7	91.1	42.9	29.7	49.4	66.3
search	43.3	55.8	28.9	44.0	56.2	27.7	24.8	44.7	80.0
sor	39.0	61.1	56.8	71.0	95.3	34.2	30.0	51.4	71.4
sparse	36.7	59.0	60.5	67.6	92.9	37.5	29.8	51.2	72.0
pseudojbb	33.7	52.5	55.8	44.2	63.0	42.4	27.6	46.3	67.9
Average			61.7			48.7			72.9

Space and Immortal Space are excluded from these measurements. We observe that the object sizes in the collector-specific space are indeed smaller than the average object size presented in Table 2.7. This is due to the array objects since for non-array objects, there is no significant difference between Tables 2.7 and 2.8. This confirms the general belief that non-arrays are small, while arrays are generally large. In general, the average object size increases by 61.7%. If we try to translate this into the impact on cache performance, we can state that for the objects in the collector-specific spaces, in terms of cache lines, an IBM POWER4 L1 D-cache line can hold 3.7 and 2.3 objects on average in 32-bit and 64-bit mode, respectively. This is a difference of more than one object per cache line on average. These measurements clearly illustrate that the object size increase problem when transferring from 32-bit systems to 64-bit systems is even worse for the majority of objects than the 45.3% increase we measured on average.

Objects allocated by the Jikes RVM

Because Jikes RVM itself is written in Java and hence its objects get intermingled with application objects, it is interesting to make a comment about the Jikes RVM and how it might influence our measurements. The data presented so far included application objects as well as objects supporting the internals of the VM. All VM-allocated data are heap objects and these objects are not separated from the application data. In previous work, authors typically only reported object sizes for objects belonging to the application and not the virtual machine because they used a VM not written in Java, see for example [26]. In order to validate if our conclusions are also valid for other VMs, not written in Java, we choose to present the data once again, distinguishing between application objects and VM objects. In order to acquire this data, we examined the call stack. When the frame of a method is encountered on the stack that can not be assigned to either the application, or the RVM, we simply walk up the stack until such a decision can be made. Examples of such undecidable methods, are library methods and some VM methods that merely assist other methods to perform certain functionality, e.g., object allocation, exception handling, etc.

Table 2.9 shows the average object size in 32-bit and 64-bit mode solely for objects allocated by the application, while Table 2.10 is showing the same data for objects allocated by Jikes RVM. As expected, the VM object sizes are much more constant across the different bench-

Table 2.9: Average Application object size (in bytes) in 32-bit and 64-bit VM mode for all objects, array objects and non-array objects.

	overall			array objects			non-array objects		
Benchmark	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)
db	24.8	47.4	91.2	158.7	289.2	82.2	16.4	32.3	96.6
jack	37.9	58.8	55.3	50.5	74.7	47.8	25.5	43.2	69.7
javac	31.8	50.9	60.0	47.6	68.0	42.9	24.6	43.0	75.2
jess	35.0	60.0	71.7	48.7	97.1	99.2	28.0	41.5	47.8
antlr	54.9	70.9	29.2	70.0	85.6	22.3	22.8	39.7	74.0
fop	40.8	60.0	47.2	61.8	84.4	36.5	28.6	46.0	60.5
hsqldb	34.8	65.2	87.3	48.6	109.0	124.0	29.6	48.7	64.5
pmd	23.9	42.7	78.9	52.1	97.0	86.1	19.2	33.7	75.6
crypt	1,724,320.0	1,724,350.0	0.0	4,839,190.0	4,839,230.0	0.0	23.6	44.6	89.4
heapsort	1,219,610.0	1,219,630.0	0.0	3,846,400.0	3,846,440.0	0.0	23.2	43.9	89.1
lufact	15,733.8	15,738.9	0.0	16,159.2	16,163.9	0.0	24.7	45.9	85.7
moldyn	83.6	96.2	15.2	452.7	829.4	83.2	79.1	87.3	10.4
search	44.2	56.2	27.2	44.2	56.2	27.2	24.4	45.7	87.5
sor	15,729.3	15,734.4	0.0	16,163.0	16,167.7	0.0	23.1	43.9	90.2
sparse	533,483.0	533,509.0	0.0	1,412,120.0	1,412,160.0	0.0	23.4	44.5	89.9
pseudojbb	40.5	60.4	49.2	84.8	111.4	31.3	27.5	45.4	65.1
Average			38.3			42.7			73.2

Table 2.10: Average Jikes RVM object size (in bytes) in 32-bit and 64-bit VM mode for all objects, array objects and non-array objects.

	overall			array objects			non-array objects		
Benchmark	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)	32-bit	64-bit	increase (%)
db	32.6	58.4	79.3	56.0	100.3	79.0	28.6	51.3	79.5
jack	31.3	56.3	80.1	56.1	92.0	63.9	26.6	50.3	89.3
javac	37.0	64.5	74.5	78.5	129.3	64.7	28.6	51.6	80.3
jess	31.9	57.1	78.9	53.8	94.4	75.5	27.6	49.9	80.6
antlr	33.4	56.8	70.0	44.3	68.4	54.3	29.0	51.9	79.1
fop	87.8	115.9	32.0	181.0	217.5	20.1	27.5	47.5	72.5
hsqldb	45.7	61.5	34.6	133.1	113.8	-14.5	28.4	50.8	79.0
pmd	34.1	58.8	72.6	61.2	94.2	54.0	27.8	50.4	81.5
crypt	42.9	79.9	86.2	108.0	199.0	84.3	30.8	54.3	76.1
heapsort	37.3	67.0	79.5	80.2	147.3	83.7	27.4	48.3	76.6
lufact	40.0	71.4	78.3	91.9	167.7	82.4	28.2	49.4	75.5
moldyn	43.1	72.1	67.3	93.2	151.1	62.2	27.8	48.0	72.4
search	30.5	55.3	81.5	59.1	108.6	83.6	24.8	44.7	80.0
sor	49.2	86.6	76.1	117.4	210.8	79.5	30.0	51.5	71.3
sparse	44.3	77.5	75.1	108.4	192.6	77.7	29.8	51.2	72.0
pseudojbb	21.5	38.2	77.2	17.8	31.5	77.6	28.0	50.4	79.9
Average			71.5			64.3			77.8

marks than the application objects, especially the non-array object sizes seem to be nearly constant across all benchmarks. An observation that can be made from both tables is that the VM objects are much more sensitive to the size increase due to 64-bit execution. For non-array VM objects, on average there is a 77.8% size increase, which is slightly larger than the 73.2% increase for non-array application objects. For array objects the contrast is much bigger: 64.3% size increase for VM arrays against only 42.7% increase on average for application arrays. This can be explained by the fact that the VM uses lots of arrays for housekeeping purposes and many of those arrays contain references. In general, Jikes RVM objects have a 71.5% object size increase when going from a 32-bit VM to a 64-bit VM, while the application data only suffers a 38.3% increase on average. In general when we compare Table 2.7 with Table 2.9 we can conclude that the data we measured for the application objects are more or less comparable to the data we measured for both VM and application objects. This gives us confidence that the intrusion of Jikes RVM objects is limited and that the results obtained here will be in line with results for VMs not written in Java.

The object size increase due to the memory manager

As discussed before, the average overall object size increases with 45.3% measured with the GenMS collector. We already identified the four main causes for this object size increase, namely: (i) increased pointer size, (ii) the increased header, (iii) the increased intra-object alignment and (iv) the increased inter-object alignment. While the first three causes are memory manager independent, the last one, however, is not. In this section, we would like to quantify how much of this increase is dependent on the memory manager used. The additional memory overhead introduced by the memory manager can be further split up into two parts, but both are a consequence of the difference in object alignment between 64-bit mode and 32-bit mode. The 64-bit VM will align all objects at an 8-byte boundary, while the 32-bit VM will align most objects at a 4-byte boundary. The first part is inter-object alignment. Inter-object alignment comes from aligning the object pointer in the allocated heap space. The second part has to do with fixed-sized cells. Some memory managers use fixed-sized cells for allocating objects (in Jikes RVM, all spaces that use a MarkSweep strategy). When the object size does not match the cell size used, an additional overhead incurs, i.e., a number of bytes remains unused in

Table 2.11: Raw object size, object size after inter-object alignment and total object heap size in 32-bit and 64-bit mode for the MarkSweep collector. The right most columns shows the average increase for each stage when going from 32-bit mode to 64-bit mode.

	32-bit			64-bit			Increase (%)		
Benchmark	raw object size	+inter object alignment	on heap	raw object size	+inter object alignment	on heap	raw object size	+inter object alignment	on heap
db	25.5	25.5	25.9	48.1	52.1	55.9	88.4	104.0	115.6
jack	37.1	37.3	39.3	56.2	60.2	62.2	51.5	61.5	58.3
javac	33.0	33.0	35.4	52.5	56.5	60.0	59.3	71.2	69.3
jess	34.3	34.9	38.2	58.5	62.5	65.7	70.5	79.1	71.9
antlr	44.8	45.1	51.6	60.9	64.9	72.5	35.9	44.1	40.6
fop	63.7	63.9	69.9	84.9	88.9	95.4	33.2	39.1	36.4
hsqldb	37.2	37.6	40.4	62.9	66.9	70.8	69.0	78.0	75.2
pmd	24.1	24.1	27.4	42.9	46.9	51.9	77.7	94.2	89.9
crypt	731.4	731.8	734.4	891.6	895.6	900.2	21.9	22.4	22.6
heapsort	374.1	374.3	376.7	405.5	409.5	413.8	8.4	9.4	9.9
lufact	171.7	172.0	177.3	202.0	206.2	213.1	17.7	19.9	20.2
moldyn	43.6	44.0	47.1	71.4	75.4	79.9	63.7	71.5	69.7
search	43.5	43.5	47.4	56.1	60.1	64.1	28.9	38.1	35.2
sor	287.3	287.9	296.4	331.0	335.3	344.9	15.2	16.5	16.4
sparse	309.5	309.9	312.7	352.5	356.5	361.0	13.9	15.0	15.4
pseudojbb	33.7	34.2	35.8	50.8	54.8	57.7	50.5	60.4	61.1
Average							44.1	51.5	50.5

a cell (internal fragmentation). The cell size in the 64-bit VM, is also a multiple of 8 bytes.

To quantify the heap overhead introduced by the memory manager, we first consider the MarkSweep collector in Jikes RVM because it is the collector with the worst overhead of all 4 collectors used. The MarkSweep collector in Jikes RVM uses fixed-sized cells per page and a data structure to keep track of used and unused cells per page. The results are shown in Table 2.11: the raw object size, the additional overhead due to inter-object alignment and the additional overhead due to the memory manager using fixed cells. The average increase in object size when going from 32-bit mode to 64-bit mode with the MarkSweep collector is 50.5%. When looking at the raw object size only (i.e., without the overhead of the memory manager), the average increase is only 44.1%. Since this number is lower than after alignment, it is clear that the extra alignment overhead introduced by the memory manager is much worse on 64-bit platforms than on 32-bit platforms. This is not surprising since the 32-bit VM tries to align most of the time to a 4-byte boundary, while the 64-bit VM exclusively aligns on an 8-byte boundary.

As said before, the MarkSweep collector has the worst overhead of all 4 collectors used. The extra allocation overhead of the MarkSweep collector is 2.6% and 5.0% on average for 32-bit and 64-bit mode, respectively, while it is only 0.8% and 1.3% on 32-bit and 64-bit mode for the SemiSpace collector, respectively. Remark that for the generational collectors, the memory manager overhead actually is not constant over time, because the generational collector copies objects from the nursery to the mature generation. The measurements for the generational collectors were done during object allocation in the nursery generation and by consequence do not include the (changed) overhead for objects moved from the nursery to the mature generation. As such their measured overhead is the same as with the SemiSpace collector (same memory strategy used in nursery generation as in one half of the SemiSpace collector). For GenCopy, both generations have the same allocation policy, so the overhead will remain the same. For GenMS, however, in each of the generations a different collector is used: the nursery uses the SemiSpace collection strategy whereas the mature generation uses MarkSweep. As such, the real average overhead of the GenMS collector lies somewhere between the overhead of the SemiSpace collector and the overhead of the MarkSweep collector.

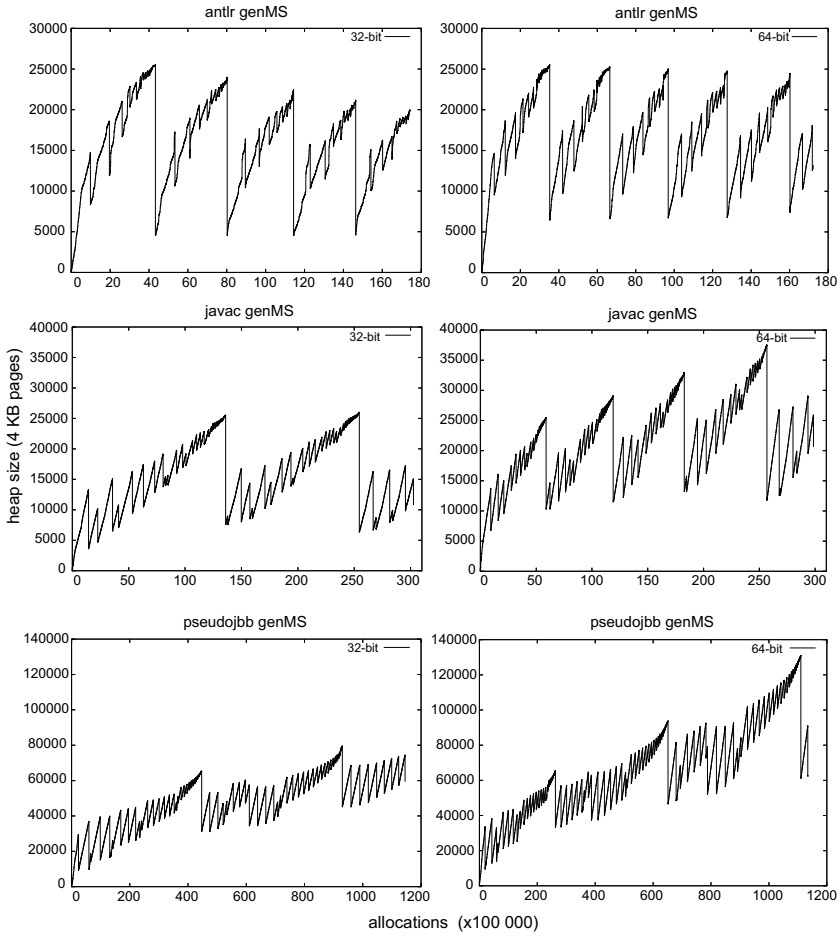

Figure 2.2: Heap growth for GenMS collector as a function of time (measured per allocation site) for three benchmarks, antlr (top), javac (middle) and pseudojbb(bottom) for 32-bit processing (left column) and 64-bit processing (right column).

2.4.2 Run time behavior of the heap

Until now we studied the size of objects at allocation time. Table 2.11 provided the average size per allocated object, i.e., the size for each allocated object is used to compute the average object heap size. Even more interesting when evaluating a VM, is not the object size *per se*, but the amount of memory it occupies at run time. At run time, the heap only contains live objects, so by consequence, the actual run time

Table 2.12: Number of minor and major GCs under the GenMS and GenCopy collection scheme for the 32-bit and 64-bit scenario.

	GenMS				GenCopy			
	minor GCs		major GCs		minor GCs		major GCs	
Benchmark	32 bit	64 bit	32 bit	64 bit	32 bit	64 bit	32 bit	64 bit
db	22.9	62.7	1.0	2.0	18.1	32.8	1.0	3.0
jack	39.1	184.0	0.0	2.0	54.8	193.9	0.1	2.1
javac	83.2	120.7	2.0	3.5	82.6	115.1	3.0	5.9
jess	28.6	75.9	0.0	0.0	31.8	167.5	0.0	1.0
antlr	68.1	106.7	4.0	5.0	68.2	104.7	5.0	5.9
fop	5.5	25.3	0.0	0.5	20.5	15.2	0.9	1.0
hsqldb	35.6	58.3	3.0	5.0	26.6	60.1	4.0	6.0
pmd	260.6	275.6	8.0	13.0	230.7	214.7	9.0	13.9
crypt	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
heapsort	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
lufact	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
moldyn	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
search	13.8	18.9	0.0	0.0	14.1	21.0	0.0	0.0
sor	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
sparse	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
pseudojbb	100.3	116.9	3.0	4.1	91.0	112.2	4.7	6.2

heap, when comparing 64-bit with 32-bit computing, might grow more or less than the average 45.3% object size increase reported above.

The Java heap grows until its size reaches a given threshold, after which a GC occurs. A GC tries to shrink down the heap to a lower size; at the same time the threshold might increase. The next time a GC occurs, the heap might grow larger than with the previous GC, because the GC threshold value is dynamically adjusted—but never exceeds the maximum heap size. We now take a look at the growing/shrinking behavior of the heap as a function of time. In Figure 2.2, the heap size is shown as a function of time measured by the number of allocations for the GenMS collector. This is shown for three benchmarks, antlr, javac and pseudojbb. Graphs for the other collectors are similar. In most cases, the high watermark heap size in 64-bit mode is only slightly bigger than in 32-bit mode, however, in some cases, for example for pseudojbb, we observe that the 64-bit VM uses nearly twice as much heap size as the 32-bit VM.

In general, Figure 2.2 shows similar growing/shrinking behavior for the 32- and 64-bit VMs. Each larger dip represents a major collection. These graphs show that the major collections are performed more

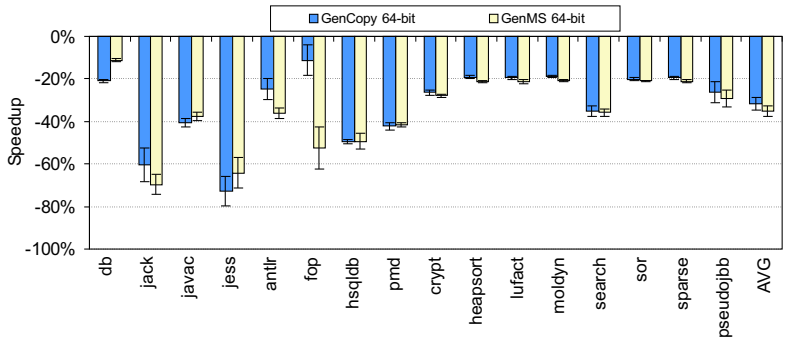

Figure 2.3: Garbage collection performance: 64-bit mode versus 32-bit mode.

frequently in 64-bit mode. For example, for javac, 2 large dips are observed in 32-bit mode, whereas in 64-bit mode 4 large dips can be recognized. Table 2.12 quantifies the actual number of garbage collections performed for the GenMS and GenCopy collectors. As expected, the number of GCs increases for the 64-bit VM. The number of minor and major GCs increases by on average 60.1% and 64.8%, respectively.

Now in order to quantify the overhead of the memory increase due to 64-bit mode on the memory system, we measure the time spent during garbage collection. Figure 2.3 shows a slowdown in GC time when going from 32-bit mode to 64-bit mode. This graph shows GC speedup, hence the negative values. We observe that about half the benchmarks show about 20% slowdown for the garbage collector. Some benchmarks have over a 50% slowdown in terms of GC performance: for the GenMS collector this is the case for jack (69.6%), jess (64.3%) and fop (52.4%); for the GenCopy collector this is the case for jess (72.6%) and jack (60.4%). On average we observe a 33.4% slowdown in terms of GC performance for both collectors.

So far, we did not take object lifetime into account. The longer an object is reachable, the longer it occupies memory and, in the case of 64-bit mode, the worse the extra memory overhead might be compared to 32-bit mode. For example, a long-lived object with a large memory overhead in 64-bit mode might set more pressure on the performance of 64-bit mode than a short-lived object. This can be explained by the fact that the memory space a short-lived object occupies, can soon be reclaimed by the garbage collector, while a long-lived object occupies memory space typically across multiple garbage collections. In order

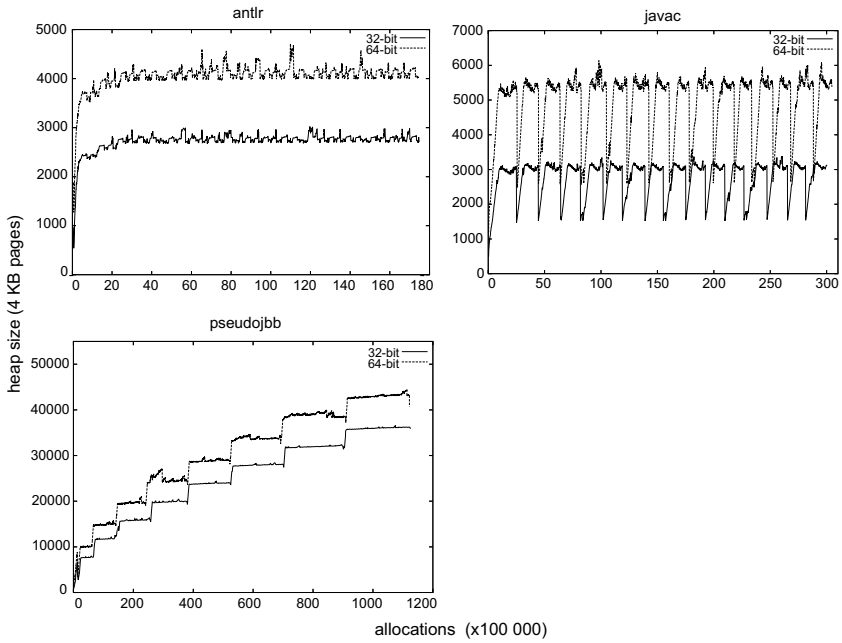

Figure 2.4: Maximum reachable bytes (measured in 4KB pages) as a function of time (measured per allocation site) for three benchmarks, antlr (top left), javac (top right), pseudojbb (bottom) for 32-bit and 64-bit processing.

to verify that not only the short-lived objects introduce extra memory overhead, we measure the maximum reachable bytes of the application at runtime[8]. The maximum reachable bytes is a good indication for the minimum heap size an application needs since these are all occupied by objects that can not be reclaimed by the garbage collector: it will contain almost exclusively long-lived objects. Figure 2.4 shows the maximum reachable bytes (in 4KB pages) for the antlr, javac and pseudojbb benchmark, measured at an interval of 0.5 MB allocated bytes. These graphs show that the extra memory overhead in terms of maximum reachable bytes is about 46% for antlr, 78% for javac and only 20% for pseudojbb when comparing 64-bit mode to 32-bit mode. If we compare these numbers with Table 2.7, we see that these overheads are even worse for antlr and javac than the average object size increase, measured for all objects. For pseudojbb it is less: 20% compared to 55.8%. Pseudojbb is a server-side benchmark, that generates more small short-lived objects.

[8] Another possible experiment that takes an object's lifetime into account is to measure the (space x time) product.

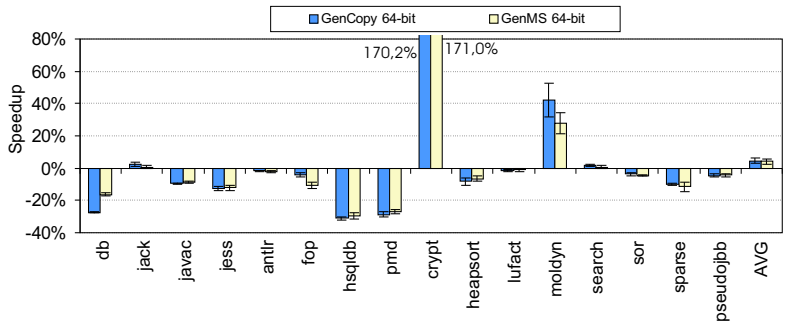

Figure 2.5: Overall speedup for 64-bit mode compared to 32-bit mode.

2.5 Overall Performance

In this section, we quantify the impact on execution time when comparing 64-bit computing with 32-bit computing. We then explain the measured differences by discussing other metrics such as (i) the number of instructions executed, (ii) the number of cache misses and (iii) the number of TLB misses. This is done using the hardware performance monitors. In these measurements we use the GenCopy and GenMS collectors of Jikes RVM, because generational collectors perform better in general and are used in most production VMs.

2.5.1 Execution time

Figure 2.5 shows the speedup for 64-bit mode compared to 32-bit mode. Negative values thus indicate that 64-bit mode is slower than 32-bit mode execution. In general, 64-bit mode is slower than 32-bit mode. For some benchmarks, we observe large performance decreases of up to 31.1% (hsqldb, pmd and db under the GenCopy collector), For other benchmarks, 64-bit is much faster than 32-bit computing, see for example crypt (171.0%) and moldyn (42.1%). However, nearly half the benchmarks show about 5% to 10% performance degradation running in 64-bit mode, while the remaining other benchmarks are barely affected.

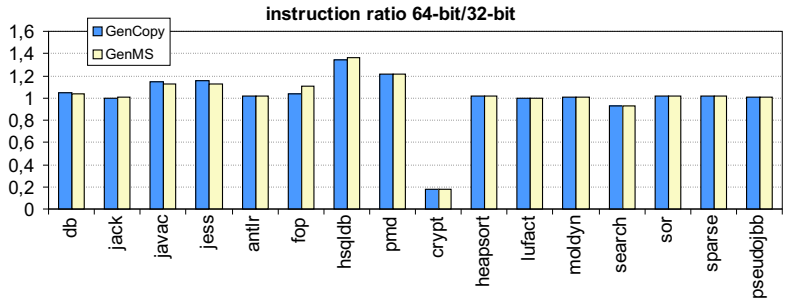

Figure 2.6: Ratio of the number of executed instructions of 64-bit to 32-bit mode.

2.5.2 Number of instructions executed

Figure 2.6 quantifies the ratio of executed instructions in 64-bit mode versus 32-bit mode. For most benchmarks, this ratio is close to 1, with a slight increase for most benchmarks in 64-bit mode. There are two benchmarks with a high increase in executed instructions, hsqldb (36.6 %) and pmd (21.4 %). These extra instruction executed in 64-bit mode come from e.g., extra sign operations on 32-bit values. The significantly higher dynamic instruction count explains the worse performance under 64-bit mode for hsqldb and pmd. Two other benchmarks have a significantly lower dynamic instruction count under 64-bit mode than under 32-bit mode: crypt (-83 %) and search (-8 %). From a detailed analysis we discovered that crypt and search perform a large number of arithmetic operations on `longs`. As such, they can benefit from the 64-bit instructions available in 64-bit mode.

2.5.3 Data cache misses

In this section as well as in the next subsection we will focus on the memory system performance of the data stream only. We do not consider the instruction stream because a larger variability between 64-bit and 32-bit processing was observed in the data stream than in the instruction stream. We first study the performance of the data caches; in the next subsection we discuss data translation lookaside buffer (D-TLB) behavior. Figures 2.7, 2.8 and 2.9 show the numbers of D-cache misses per 1000 instructions in the reference run for the L1, L2 and L3 D-cache, respectively. The 32-bit runs were used as reference runs, for

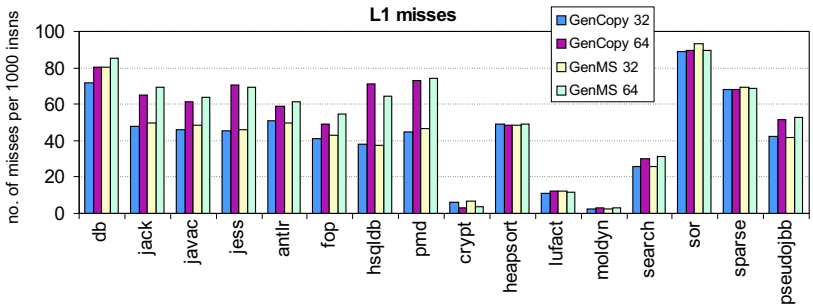

Figure 2.7: The number of L1 D-cache misses for 64-bit and 32-bit mode, per 1000 executed instructions in 32-bit mode.

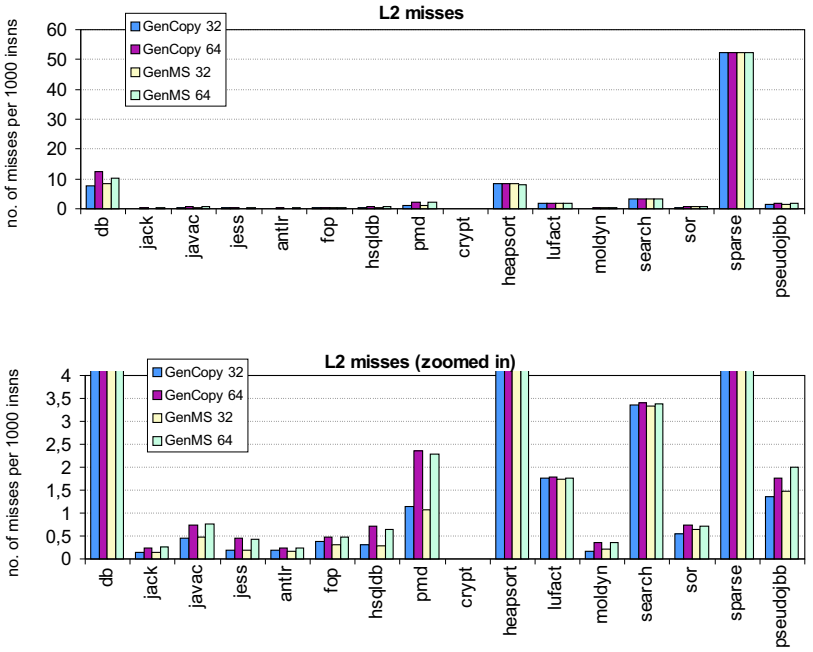

Figure 2.8: The number of L2 D-cache misses for 64-bit and 32-bit mode, per 1000 executed instructions in 32-bit mode. The only difference between the first and second graph is the scale of the vertical-axis.

both 32-bit and 64-bit modes, so that the difference in dynamic instruction count is not reflected in these measurements. The graphs for the L2 and L3 D-cache in Figures 2.8 and 2.9 are shown a second time to

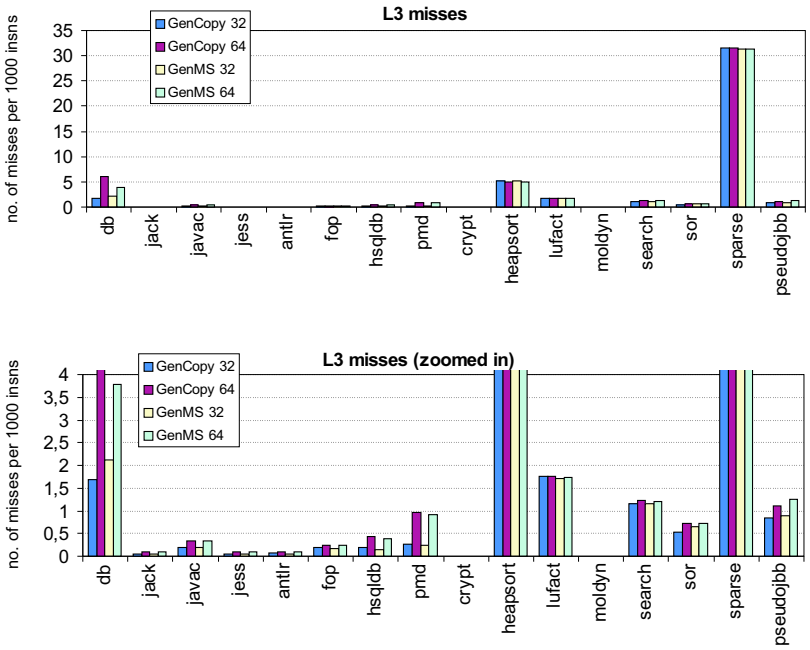

Figure 2.9: The number of L3 D-cache misses for 64-bit and 32-bit mode, per 1000 executed instructions in 32-bit mode. The only difference between the first and second graph is the scale of the vertical-axis.

magnify the results for benchmarks with a small number of misses per 1000 instructions. with a small number of misses per 1000 instructions.

Given the increased object size, the increased alignment, and by consequence an increased heap size, we expect that 64-bit VMs will have an increased number of D-cache misses over 32-bit VMs. Figures 2.7, 2.8 and 2.9 show that most benchmarks indeed experience an increase in the number of misses. On average the increase is 21.3%, 48.8% and 66.5% for the L1, L2 and L3 caches, respectively. Observe that the 3 benchmarks with the largest performance penalty (db, hsqldb and pmd) in Figure 2.5, correspond with the 3 benchmarks showing the largest increase in L3 misses. Note that most Java Grande Forum benchmarks have nearly the same number of misses in 64-bit mode as in 32-bit mode on L2 and L3 cache levels, and a modest increase in number of L1 D-cache misses. The SPECjvm98, Pseudojbb and DaCapo benchmarks on the other hand, generally have larger cache miss increases on all cache levels, although most of them have a very low

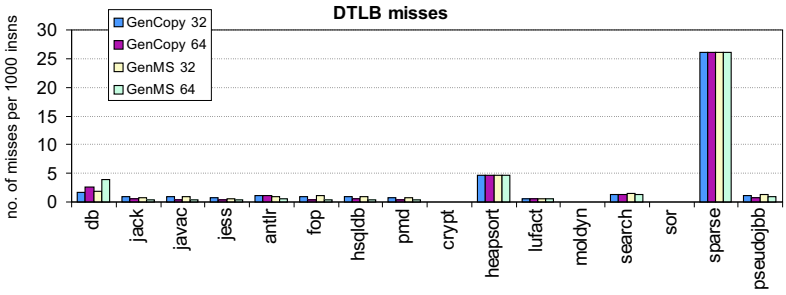

Figure 2.10: Number of D-TLB misses for 64-bit and 32-bit mode, per 1000 executed instructions in 32-bit mode

number of misses at the L2 and L3 levels. This can be explained by the fact that the JGF benchmarks show relatively small object size increases compared to SPECjvm98 between 64-bit mode and 32-bit mode (see Table 2.7). As we discussed earlier, JGF benchmarks have large numeric data structures and SPECjvm98 benchmarks have more pointer-rich data structures. Note that especially SPECjvm98's `db` suffers in terms of L3 cache misses in 64-bit mode: an increase of more than a factor 3.5 in the number of misses. This can be explained in part by the fact that `db` experiences the highest heap object size increase (89.6%) as reported in Table 2.7. This high increase in data cache misses explains the high performance degradation for `db` (37.9%) as observed when comparing 64-bit computing to 32-bit computing.

2.5.4 D-TLB performance

Figure 2.10 shows the number of D-TLB misses per 1000 instructions in 32-bit mode, for both 64-bit mode and 32-bit mode. Due to the larger object sizes in 64-bit mode, we expect more pages will get accessed, and thus we expect more TLB misses due to an increase in the number of TLB conflicts. We observe that the number of D-TLB misses for the Jikes RVM only increases for `db`. The number of D-TLB misses remains constant for a small set of benchmarks, while for several benchmarks we observe a decrease. On average we observe a(n) (unexpected) decrease in D-TLB misses of 25.1% in 64-mode compared to 32-bit mode. This is not a peculiarity of Jikes RVM, since a decrease in D-TLB misses was also found for the production IBM 1.4.0 VM in [69].

2.6 Related work

As mentioned in the introduction, the increased memory usage caused by using 64-bit pointers is not unexpected. We postpone the description of the related works addressing the increased memory usage in 64-bit VMs to the following chapters.

To the best of our knowledge there is no prior work on comparing 64-bit Java workloads with 32-bit Java workloads. However, several studies have been done on characterizing the memory allocation behavior and memory system performance of such workloads. All these studies were done on one particular platform, either 32-bit or 64-bit—and, as far as we can verify this statement, most (if not all) of them are done on a 32-bit platform.

The study that is probably most related to this work, is the work done by Dieckmann and Hölzle [26] in which they characterize the allocation behavior of SPECjvm98 benchmarks. They measure the heap size as a function of time. They quantify heap composition, i.e., the differentiation between array and non-array objects on the heap. And they also compute object size and object alignment. This study was done on a 32-bit platform.

Shuf et al. [64] characterize the memory behavior of Java workloads. They measure for example the distribution of heap accesses over different types such as object fields, arrays and virtual method tables. In addition, they also measure cache miss rates and TLB miss rates.

Blackburn et al. [12] present a detailed study on the performance impact of garbage collection. For this use they use hardware performance monitors on three different hardware platforms.

Li et al. [52] use complete system simulation to study the SPECjvm98 benchmarks. They conclude that most of the kernel activity is due to TLB miss handling and that those TLB misses are due to JIT compilation, garbage collection and class loading.

Kim and Hsu [45] characterize the memory system behavior of Java workloads. They measure the lifetime characteristics of objects, the temporal locality and the impact of associativity on cache miss rate.

Next to these there exist yet other studies characterizing the (memory) performance of Java workloads, more specifically the time varying behavior in terms of cache miss rates and TLB miss rates [66], method-level phase behavior [30], impact of VMs and input sets on overall Java performance [28], Java middleware benchmarks (SPECjbb2000

and SPECjAppServer2001) using real hardware as well as full-system simulation [43], Java TPC-W which exercises the web server and transaction processing of a typical e-commerce web site [16], cache behavior of SPECjvm98 benchmarks [59], Java server applications (SPECjbb2000 and VolanoMark 2.1.2) [54, 61].

2.7 Conclusion

The purpose of this chapter was to compare 64-bit VMs with 32-bit VMs for Java applications in general and the allocation behavior and memory system performance more in particular, so that we have a firm basis to build upon in subsequent chapters when attacking the cost of the increased memory usage of 64-bit computing.

We performed our study on a collection of different benchmarks suites (SPECjvm98, DaCapo, pseudojbb and Java Grande Forum) on the Jikes Research VM. The underlying hardware platform was the 64-bit PowerPC-based IBM POWER4 processor. By running virtual machines both in 32-bit and 64-bit mode, we were able to compare the characteristics and performance of 32-bit to 64-bit virtual machines for Java.

We measured and compared the average object size, and the heap growth from 32-bit to 64-bit computing. We conclude that the average object size increases by 45.3% in 64-bit mode compared to 32-bit mode. This is due to the increased pointer size (64 bits versus 32 bits), the increased header and the increased alignment. We have shown that this leads to an increased number of garbage collections performed and to an increased amount of time spent during GC when run in 64-bit mode. For the setup used in this chapter, a Java application running on a 64-bit VM shows an average GC slowdown of 33.4% compared to running on a 32-bit VM. Also, the number of minor and major collections are increased by 60.1% and 64.8%, respectively.

Using the POWER4's hardware performance monitors, we were able to measure the total execution time, the number of instructions executed, the number of data cache and TLB misses. We conclude that 64-bit computing inside Jikes RVM is generally a few percentage slower than 32-bit computing for most benchmarks. Some benchmarks are much faster because of the extra 64-bit instructions available in 64-bit mode. We conclude that 64-bit Java results in a larger number of

data cache misses at all levels in the cache hierarchy: 21.9%, 50.3% and 66.2% more misses for the L1, L2 and L3 caches, respectively.

Chapter 3

Object-Relative Addressing

To live a creative life, we must lose our fear of being wrong.
Joseph Chilton Pearce

The previous chapter identified the increased pointer size as the main reason why 64-bit Java environments demand about 45.3% more memory than 32-bit environments. In this chapter we examine a pointer compression technique, called Object-Relative Addressing (ORA), to reduce the memory usage of 64-bit pointers in the context of Java virtual machines. Unlike previous work on the subject, like that of by Adl-Tabatabai et al. [3] which targeted applications that need less than 4 GB of memory, our compression technique allows for applying pointer compression to Java programs that allocate more than 4 GB of memory. The experimental results evaluating ORA show that the overhead introduced is statistically insignificant on average compared to the raw 64-bit pointer representation, while reducing the total memory demands by on average 10.7%, 12.2% and 11.3% for the DaCapo, SPECjvm98 and Pseudojbb benchmark suites, respectively.

3.1 Introduction

Object-Relative Addressing (ORA) is a pointer compression technique with a simple basic idea. It compresses pointers in object fields as 32-bit offsets relative to the referencing object's address. The 64-bit virtual address of the referenced object is then obtained by adding the 32-bit offset to the 64-bit virtual address of the referencing object.

Compressing pointers as a relative offset introduces a number of issues that need to be dealt with. For instance, at store time, we need to

check if a pointer is at all compressible, and, at load time, the pointer needs to be decompressed properly. Moreover a fall-back mechanism is needed in case a pointer value is not compressible. Another issue is the the null pointer. It will need a new compressed 32 bit representation: null pointers will need to be initialized to this new representation instead of using all zeros. Also extra care needs to be taken to make sure all pointers are updated correctly during garbage collection: the relative 32-bit offset gets invalid in both situations where the referenced or the referencing object is moved.

Our experimental results on Jikes RVM using the SPECjvm98, Java Grande Forum, SPECjbb2000 and the DaCapo benchmarks on an IBM POWER4 machine show that object-relative addressing does almost not incur any run time overhead. On average, no statistically significant performance impact is observed for the GenMS collector; the GenCopy collector shows a small performance degradation of 1.5%. The benefit of ORA comes in terms of memory savings: the amount of allocated memory gets reduced by more than 10% for several benchmark suites and, in addition, ORA reduces the number of minor and major GCs with 10.6% and 17.8% on average, respectively.

We envision that object-relative addressing is to be used in conjunction with a memory management strategy that strives at limiting the number of inter-object references that cross the 32-bit address range. Crossing the 32-bit address range might lead to pointers that can not be compressed. This incurs overhead because then extra data structures need to be accessed for retrieving the uncompressed 64-bit pointer. Limiting the number of incompressible pointers thus calls for a memory allocator and garbage collector that strives at allocating objects within a virtual memory region that is reachable through the (signed) 32-bit offset. Such memory allocators and garbage collectors can be built using techniques similar to object collocation [34], connectivity-based memory allocation and collection [38, 39], region-based systems [22, 57], etc.

3.2 Object-Relative Addressing

The prior work on the subject by Adl-Tabatabai et al. [3] propose a straightforward compression scheme for addressing the increased memory usage in 64-bit Java virtual machines. They represent 64-bit pointers as 32-bit offsets from a fixed base address of a contiguous

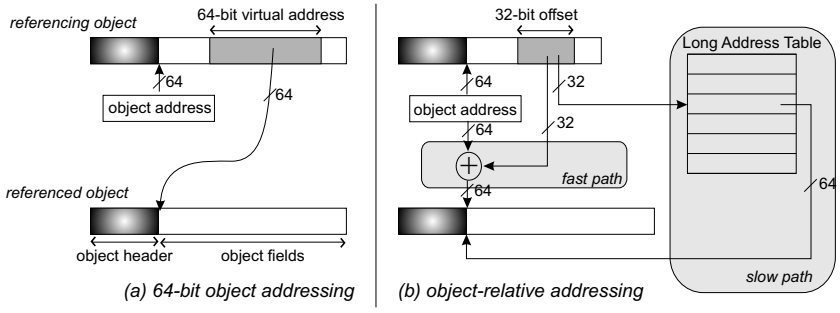

Figure 3.1: Illustrating the basic idea of object-relative addressing (on the right) compared to the traditional 64-bit addressing (on the left).

memory region. Decompressing a pointer then involves adding the 32-bit offset to this fixed base address, compressing a 64-bit pointer results in only storing the 32 least significant bits. The fact that 64-bit virtual addresses are represented as 32-bit offsets from a fixed base address implies that this pointer compression technique is limited to Java programs that demand less than 4 GB of memory.

Object-Relative Addressing (ORA) is a pointer compression technique for 64-bit Java virtual machines that does not suffer from the 4 GB heap limitation in Adl-Tabatabai et al.'s method [3]. The goal of ORA is to enable heap pointer compression for all Java programs, even for programs that allocate more than 4 GB of memory.

3.2.1 Basic idea

Figure 3.1 illustrates the basic idea of object-relative addressing (ORA) and compares ORA to the traditional way of referencing objects in 64-bit Java virtual machines. We call the referencing object the object that contains a pointer in its data fields. The object being referenced is called the referenced object. ORA references objects through 32-bit offsets. The 'fast' decompression path then adds this 32-bit offset to the referencing object's virtual address for obtaining the virtual address of the referenced object. This is the case when both the referencing object and the referenced object are close enough to each other so that a 32-bit offset is sufficiently large. In case both objects are further away from each other in memory than what can be addressed through a 32-bit offset, ORA follows the 'slow' decompression path. In this case, the least significant bit of the 32-bit offset is set to one and the offset is then con-

```
read 32-bit object reference;
if (least significant bit
  of 32-bit reference is NOT set) {
    /* fast decompression path */
    add 32-bit object reference to 64-bit object
      virtual address to form 64-bit object address;
}
else {
    /* slow decompression path */
    index LAT for reading 64-bit object address;
}
```

Figure 3.2: High-level pseudocode for decompressing 32-bit object references.

sidered as an index into the *Long Address Table (LAT)* which holds 64-bit virtual addresses corresponding to 32-bit indices.

The end result of object-relative addressing is that only 32 bits of storage are required for storing object references. This saves memory compared to the traditional way of storing object references which requires 64 bits of storage. We now go through the details of how ORA can be implemented. We discuss (i) how pointers are decompressed, (ii) how to compress pointers, (iii) how to deal with null pointer representation, (iv) how to manage the LAT, (v) what the implications are for garbage collection, (vi) how ORA compares to Adl-Tabatabai et al.'s method [3] in terms of anticipated run time overhead and finally (vii) what the implications are for memory management,

3.2.2 Decompressing pointers

Decompressing 32-bit object references requires determining whether the fast or slow path is to be taken. This is done at run time by inspecting the least significant bit of the 32-bit offset; in case the least significant bit is zero, the fast path is taken; otherwise, the slow path is taken. This is illustrated in Figure 3.2 showing the high-level pseudocode for decompressing 32-bit object references into 64-bit virtual addresses.

The way how the high-level pseudocode is translated into native machine instructions has a significant impact on overall performance. And in addition, efficient pointer decompression is likely to result in different implementations on different ISAs. For example, in case predicated execution is available in the ISA [55], a potential implementation could predicate the fast and slow paths. Or, in case a 'base plus index

```
                              ;; R4 :referencing object's
                              ;;  virtual address
       ld4  R1, [R4 + offset] ;; load 32-bit object offset
                              ;;  and sign-extend into R1
                              ;; fast decompression path

       add  R2, R4, R1        ;; compute 64-bit address
       tst  R1, 1             ;; test least significant bit
       bre  L2                ;; if non-zero: jump to L2

  L1:  ...                    ;; referenced object's
                              ;;  virtual address is
       ...                        in R2 here

  L2:                         ;; slow decompression path
       mask R1                ;; compute LAT index
       ld8  R2, [R5 + R1]     ;; load address from LAT
                              ;;  R5 contains LAT address
                              ;;  R1 contains LAT index
       jmp  L1
```

Figure 3.3: Low-level pseudocode for decompressing 32-bit object references: the if-then decompression approach.

plus offset' addressing mode is available in the ISA, computing the address of an object field being accessed in the referenced object could be integrated into a single memory operation, i.e., the decompression arithmetic could be combined with the field access. The referencing object's virtual address plus the 32-bit offset plus the offset of the object field in the referenced object could then be encoded in a single addressing mode.

In our experimental setup using a PowerPC machine, we were not able to implement these optimizations because the PowerPC ISA does not provide predication, nor does it support the 'base plus index plus offset' addressing mode. Instead, we consider two implementations to pointer decompression that are generally applicable across different ISAs. These two decompression implementations have different performance trade-offs which we discuss now and which will be experimentally evaluated in section 3.5.

```
                              ;; R4 contains the referencing
                              ;;  object's virtual address
      ld4  R1, [R4 + offset] ;; load 32-bit object offset
                              ;;  and sign-extend into R1
                              ;; fast decompression path
      add  R2, R4, R1         ;; compute 64-bit address
L1:   ...                     ;; referenced object's virtual
                              ;;  address is in R2 here
```

Figure 3.4: Low-level pseudocode for decompressing 32-bit object references: the patched decompression approach before code patching is applied.

If-then pointer decompression.

The if-then implementation is shown in Figure 3.3. The assembler code generated for decompressing 32-bit object references implements the corresponding high-level pseudocode by optimizing for the most common case, namely the fast path. We (speculatively) compute the virtual address of the referenced object by adding the 32-bit offset with the referencing object's virtual address. In case the least significant bit of the 32-bit offset is zero, we then continue executing instructions along the fall-through path. Only in case the least significant bit of the 32-bit offset is set, we jump to the slow path. The slow path selects a number of bits from the 32-bit offset that will serve as index into the LAT. The slow path then indexes the LAT which reads the 64-bit virtual address of the referenced object.

Patched pointer decompression.

Patched pointer decompression optimizes the common case even further by assuming that the fast path is always taken. This results in the code shown in Figure 3.4. In other words, the 32-bit offset is added to the referencing object's virtual address to obtain the referenced object's virtual address. This avoids the conditional branch as needed in the if-then decompression implementation. In case the referenced object may not be reachable using a 32-bit offset, the decompression code needs to be patched. Code patching is done at run time whenever pointer compression reveals that objects may no longer be reachable using compressed pointers, as will be discussed in the next section. The decompression code after patching is shown in Figure 3.5. Code patching replaces the addition (of the 32-bit offset with the referencing object's

```
                              ;; R4 contains the referencing
                              ;;  object's virtual address
      ld4  R1, [R4 + offset] ;; load 32-bit object offset
                              ;;  and sign-extend into R1
      jmp L2

L1:   ...                     ;; referenced object's virtual
                              ;;  virtual address in R2 here
      ...
L2:   add  R2, R4, R1         ;; compute 64-bit address
      tst  R1, 0              ;; test least significant bit
      bre L1                  ;; jump to L1 in case zero
                              ;; slow decompression path
      mask R1                 ;; compute LAT index
      ld8  R2, [R5 + R1]      ;; load address from LAT
                              ;;  R5 contains LAT address
                              ;;  R1 contains LAT index
      jmp  L1
```

Figure 3.5: Low-level pseudocode for decompressing 32-bit object references: the patched decompression approach after code patching is applied.

virtual address) with a jump to a piece of code that does the pointer decompression using the if-then approach. Since most object references will follow the fast path, the patched decompression approach (before patching is applied) will be substantially faster than the if-then decompression approach.

3.2.3 Compressing pointers

Compressing 64-bit pointers to 32-bit offsets is done the other way around, see Figure 3.6. We first compute the difference between the 64-bit virtual addresses of the referenced and referencing objects. If this difference is smaller than 2 GB, i.e., can be represented by a 32-bit offset, we then store the difference as a 32-bit offset in the referencing object's data fields. If on the other hand the difference is larger than 2 GB, we allocate a LAT entry and store the referenced object's virtual address in the allocated LAT entry. The LAT entry's index is then stored in the referencing object's data fields while setting the least significant bit (LSB) of the stored LAT index. In case of the patched decompression approach, all pointer decompressions that may read the 32-bit offset need to be patched. In our experimental setup we maintain the corresponding locations in the code per object type. As such we do

```
compute difference between 64-bit virtual addresses
  of the referenced object and the referencing object;
if (difference is smaller than 2 GB) {
    /* fast compression path */
    store 32-bit offset;
}
else {
    /* slow compression path */
    allocate entry in LAT;
    store the referenced object's address in the
      LAT in allocated entry;
    store LAT index as a 32-bit value while setting
      the LSB of 32-bit value being stored;

    /* for the patched approach */
    patch pointer decompressions that need to;
}
```

Figure 3.6: High-level pseudocode for compressing 64-bit object references.

not need to patch all load locations when a too large offset is detected, but only those locations that potentially load the current object type at hand. This requires that the VM keeps track of the accesses to a given data field in an object of a given type. The patching itself is done as described in the previous section.

3.2.4 Null pointer representation

An important issue when compressing references is how to deal with null pointers. The representation of a null value in native code is typically a 64-bit zero value. Compressing a 64-bit null value to a 32-bit representation under ORA is not trivial. A naive approach would represent the compressed null value as a 32-bit zero value. However, the 32-bit null value would then be decompressed to the `this` pointer, i.e., the pointer to the object itself. This would make the null value indistinguishable from the `this` pointer.

For dealing with null pointer representation, we take the following approach. We first add the 32-bit compressed pointer to the referencing object's 64-bit virtual address. In case the least significant 32 bits of the resulting value are zero, we consider the 32-bit compressed pointer as the null value. This means we no longer have a single null value. Also special treatment is required when comparing two pointers. In

case both pointers represent the null value, a simple comparison may evaluate to not equal, for example, in case both compressed pointers come from different objects. We need to capture this special case in the virtual machine's code generator when generating code that compares pointers. In addition, given that all memory addresses with the 32 least significant bits set to zero represent null values, we cannot allocate objects at these 4 GB memory boundaries.

Another solution for the null pointer problem would be to make use of one of the free least significant bits in an aligned pointer. In the bit-stealing used under ORA, we already steal the least significant bit to determine whether the compressed value is a truly compressed pointer or an index into the LAT. Typically, 3 bits are always zero due to 8-byte alignment. So we could steal another bit to encode the null pointer. On certain architectures, e.g., Itanium, unaligned memory accesses generate traps. Under these circumstances these accesses can be flagged as null pointer exceptions. However, on our PowerPC machine, unaligned accesses are allowed and hence we are unable to implement this scheme.

3.2.5 Managing the LAT

Another important issue to deal with is how to manage the Long Address Table (LAT). Allocating LAT entries is very straightforward by advancing the LAT head pointer. Managing LAT entries is done during garbage collection. Let us first consider non-generational garbage collection. A SemiSpace garbage collector for example, which copies reachable objects from one space to the other upon a GC, requires that the LAT be recomputed, i.e., a new LAT is built up during GC and the old LAT is discarded. A Mark-Sweep garbage collector that does not need to copy reachable objects, in theory, does not require recomputing the LAT. However, in order not to let the LAT explode because of entries pointing to dead objects, a good design choice is to also recompute the LAT upon a mark-sweep collection.

For generational garbage collectors, we recommend using two LATs, one associated with the nursery generation and another one associated with the mature generation. The nursery LAT contains references in and out of the nursery space; the mature LAT contains all other references. Upon a nursery GC, all reachable nursery objects are copied to the mature generation; as such, the nursery LAT can be dis-

carded and the mature LAT possibly needs to be extended with entries for the newly copied objects. Upon a major GC, a similar strategy can be used as under a non-generational garbage collector, i.e., the mature LAT needs to be rebuilt and in addition, the nursery LAT is discarded.

In case of the unlikely event of the LAT running full — the LAT can be chosen to be sufficiently large, and, in addition, a good object allocation strategy would strive at reducing the number of LAT entries allocated — a garbage collection could be triggered to reclaim unreachable memory. The garbage collector will rebuild the LAT, and as a result the LAT will likely shrink (or if needed, the LAT size could be increased). A data structure linking memory pages makes increasing the LAT relatively easy, i.e., the LAT does not need to be copied, since it is not necessarily contiguous in memory.

3.2.6 Implications to copying garbage collectors

Object-relative addressing raises the following issue to copying garbage collectors. Consider the case where object A has a reference to object B in its data fields. Assume object A is reachable; by consequence, object B is also reachable. The garbage collector has to assume both objects are live and a copying collector will thus have to copy both objects. Assume the copying collector will first copy object A. The compressed pointer in A referencing to B then needs to be updated because object A was copied which changes the compressed pointer's base address. Upon copying object B, the compressed pointer in A referencing to B needs to be computed again because now B is moved. In other words, the compressed pointer in A needs to be recomputed twice under a copying garbage collector.

In order not to recompute the compressed pointer twice, but only once, we do the following. During garbage collection, we maintain both the original object A and a copied version of object A in the scan list, and we use the original object A to retrieve the virtual address of the referenced object B. As such, we need to recompute the compressed pointer only once, namely upon scanning object B.

3.2.7 Discussion

Note that pointer compression and decompression in ORA cannot be optimized as in the simple pointer compression technique proposed by

Adl-Tabatabai et al. [3]. Adl-Tabatabai et al. report that it is "crucial to optimize the unnecessary compression and decompression in order to get net performance gains". This can be done by considering the phase ordering between code optimization and compression/decompression arithmetics to make sure the additional compression/decompression arithmetics get optimized whenever possible. The optimizations by the Adl-Tabatabai et al. approach include for example:

- *load-store forwarding*: If a loaded 32-bit offset is subsequently stored, the 32-bit offset does not need to be decompressed and subsequently compressed again; the 32-bit offset can be stored right away. This is not the case for ORA because the base address to which the 32-bit offset relates is the virtual address of the referencing object. And since the objects from which the 32-bit offset is loaded is likely to be different from the object to which the 32-bit offset needs to be stored, the 32-bit offset to be stored needs to be recomputed.

- *reference comparison*: Comparing objects' virtual addresses can be done easily by comparing the 32-bit offsets in the Adl-Tabatabai et al. approach. This is not the case for ORA; the 64-bit virtual addresses need to be decompressed from the 32-bit offsets before allowing for a comparison, the reason being that the base addresses are likely to be different for both 32-bit compressed pointers.

- *reassociation of address expressions*: Computing the address of an object field or array element involves two additions in Adl-Tabatabai et al.'s approach: the heap base needs to be added to the 32-bit offset plus the object field's offset. Under many circumstances, one addition can be pre-computed at compile time. For example, in case of an object field access, the heap base address and the object field's offset are both constants and can be pre-computed. Again, this is an optimization that cannot be applied to ORA because the base address is not constant. A related optimization is to apply common subexpression elimination. For example, if multiple fields of the same object are accessed, then the heap base address plus the 32-bit offset is a common subexpression that can be eliminated, i.e., does not need to be recomputed over and over again. The latter optimization can also be applied under ORA.

In summary, the pointer compression approach by Adl-Tabatabai et al. allows for a number of optimizations that cannot be applied to ORA. Hence, it is to be expected that ORA will perform poorer than the pointer compression technique proposed by Adl-Tabatabai et al. However, ORA can apply pointer compression to Java programs that allocate more than 4 GB of heap memory, which cannot be done using Adl-Tabatabatai et al.'s method.

It is interesting to note that, in case the 'base plus index plus offset' memory addressing mode would be available in the host ISA — again, which is not the case in our PowerPC setup — ORA would be able to apply an important optimization that would likely close (part of) the gap between ORA and Adl-Tabatabai et al.'s technique. Pointer decompression can then be combined with field offset computation into a single address expression. In that case, the optimization done by Adl-Tabatabai et al. to pre-compute constants would be subsumed by combining the pointer decompression with field offset computation.

3.2.8 Implications for memory management

As mentioned in the introduction, object-relative addressing is envisioned to be used in conjunction with a dedicated memory management approach for allocating objects in memory regions such that all inter-object references within a memory region can be represented by a 32-bit offset. To this end, ORA can rely on previously proposed memory management approaches that allocate connected objects into memory regions while minimizing the number of references across memory regions. Example memory management approaches that serve this need are object collocation [34], connectivity-based garbage collection [38, 39] and region-based systems [22, 57]. The smarter the memory management strategy, the smaller the number of LAT accesses, the smaller the compression/decompression overhead, and thus the higher overall performance.

In this context, it is also important to note that ORA is flexible in the sense that ORA can be activated and deactivated for particular object types; or, if needed, ORA can even be activated/deactivated for particular references between pairs of object types. It was this insight on ORA's flexibility that lead us to our compression/decompression scheme with patching. The slow decompression path is not called for at the beginning of the program execution as the heap is small enough

— as such we always execute the fast path and thus eliminate executing the if-then decompression code. Once an inter-object reference is detected that cannot be represented by a 32-bit value, all the code that may possibly read the compressed pointer needs to be patched. We envision however that this is a very rare event, since there exist techniques that we can build upon, such as region-based memory managers, that never create inter-region references at all [57]. But ORA is flexible enough to handle large inter-object reference cases; namely, it uses the LAT as a safety net in case the memory management strategy would fail to allocate objects so that all pointers can be represented as 32-bit offsets.

3.3 Experimental setup

Our experimental setup is the same as for the previous chapter. We use the Jikes RVM version 2.3.5, with the GenCopy and GenMS garbage collectors[1]. The hardware platform on which we have done our experiments is the IBM POWER4. The benchmarks are taken from a variety of suites (SPECjvm98, SPECjbb2000, DaCapo and the Java Grande Forum benchmarks). In order to be able to draw statistically valid conclusions, we employ statistics to determine 95% confidence intervals from 15 measurement runs. These statistics will help us in determining whether ORA results in statistically significant or statistically insignificant performance gains or degradations. The experimental setup is described in detail in section 2.2.

3.4 Memory usage and impact on GC

We first analyze the impact of ORA on memory performance. We quantify the impact of ORA on the number of bytes allocated and the number of memory pages touched.

Figure 3.7 shows the reduction in the number of allocated bytes through object-relative addressing, compared to the 64-bit base case. Our base case is a 64-bit version of Jikes RVM which assumes 64-bit pointer representations in object data fields. Compressing 64-bit ob-

[1]The ORA technique can also be implemented for all other garbage collection schemes. We limit our implementation to the two best performing garbage collection algorithms available in Jikes RVM 2.3.5.

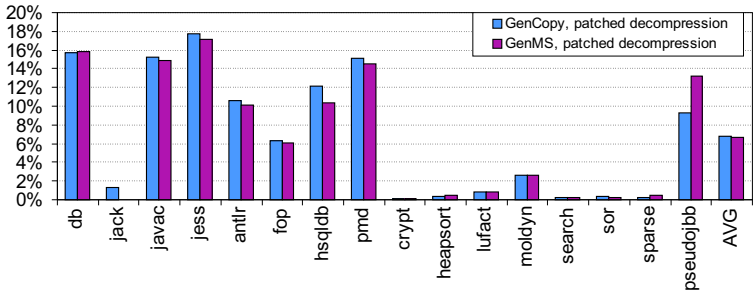

Figure 3.7: Reduction in the number of allocated bytes through ORA.

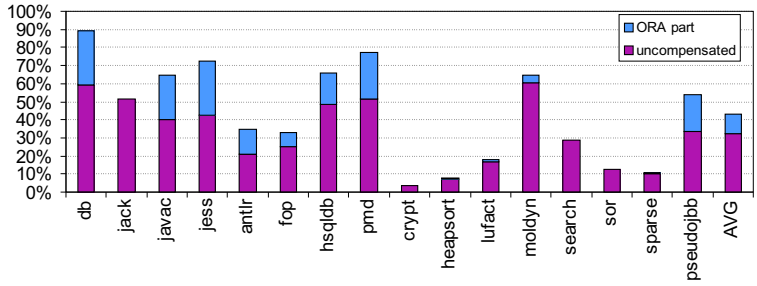

Figure 3.8: Memory usage overhead of 64-bit mode compared to 32-bit mode and the part thereof that is reduced through ORA.

ject references reduces the number of allocated bytes on average by 10.7%, 12.2% and 11.3% for the DaCapo, SPECjvm98 and Pseudojbb benchmark suites, respectively. For the Java Grande Forum benchmark suite, which operates on large numeric data structures and hence contains less references between objects, we observe an average reduction in number of allocated bytes of 0.7%. For some benchmarks it is possible that the reduction in bytes through the ORA-compression technique is so small, that it merely compensates for the extra memory usage occupied by ORA constructs (e.g., patching information). That is why we observe almost no reduction in the number of allocated bytes for jack, see Figure 3.7. To validate the effect of ORA on the reduction of the memory usage overhead introduced by the transition from 32-bit to 64-bit mode, Figure 3.8 shows the increase in number of bytes allocated when going from 32-bit mode to 64-bit mode and we mark the part that

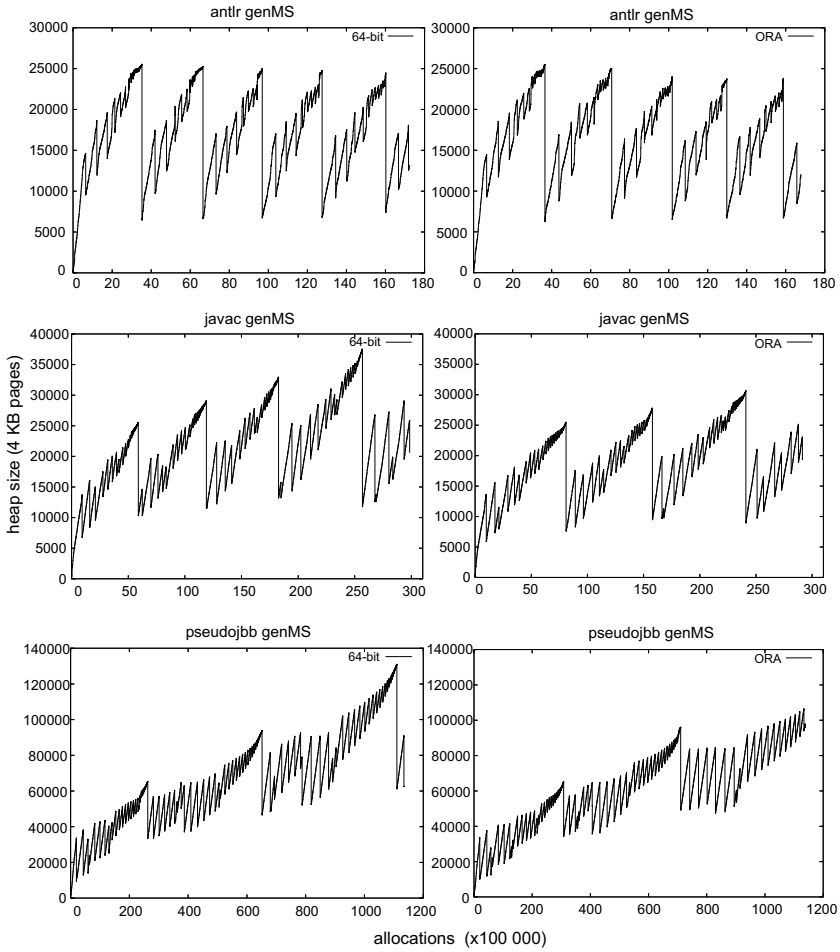

Figure 3.9: Number of pages in use as a function of time for antlr (top), javac (middle) and pseudojbb (bottom) with the GenMS collector: the base case (left column) versus ORA (right column).

the ORA technique reduces thereof. On average, more than one quarter of the memory usage overhead gets reduced.

Figure 3.9 shows the number of memory pages in use on the vertical axis as a function of time (measured in the number of allocations) on the horizontal axis for antlr, javac and pseudojbb, respectively. Each figure shows two graphs, one for the base 64-bit pointer representation (on the left), and one for the compressed pointer representation through object-relative addressing (on the right). (We observed similar curves for the other benchmarks.) The curves in these graphs increase

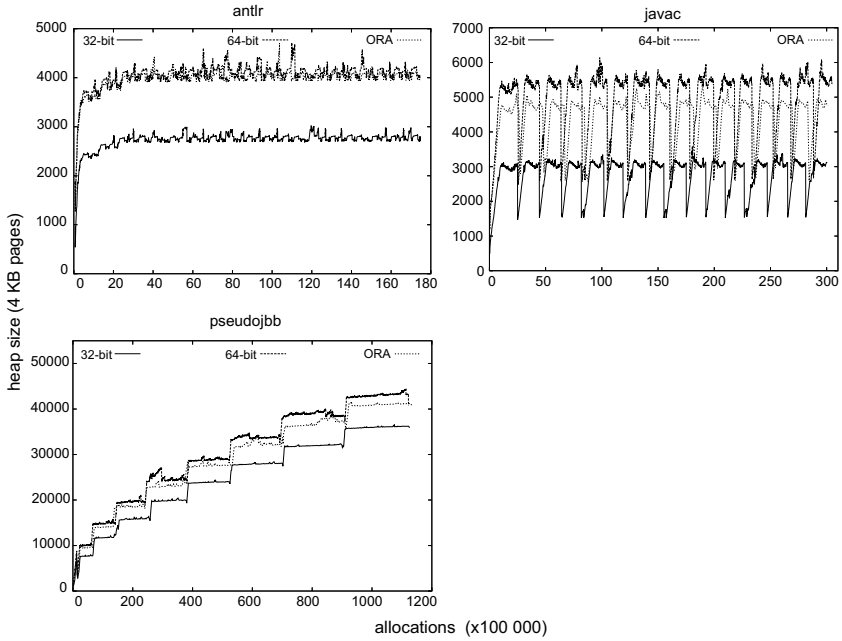

Figure 3.10: Maximum reachable bytes (measured in 4KB pages) as a function of time (measured per allocation site) for three benchmarks, antlr (top left), javac (top right), pseudojbb (bottom) for 32-bit and 64-bit processing and for ORA.

as memory gets allocated until a garbage collection is triggered after which the number of used pages drops. The small drops correspond to nursery collections; the large drops correspond to major collections collecting the full heap. The graphs for javac and pseudojbb show that the number of pages in use is lower under ORA than under the base 64-bit pointer representation. The graph also shows that the reduced number of pages in use delays garbage collections, i.e., it takes a longer time before a garbage collection is triggered.

Figure 3.10 shows the maximum reachable bytes (in pages) as a function of time (measured in the number of allocations) on the horizontal axis for antlr, javac and pseudojbb, respectively. We start from the 32-bit and 64-bit graphs from Figure 2.4, and add now the curves for the maximum reachable bytes under ORA. For antlr, we see no clear reduction in the maximum reachable bytes for ORA compared to the 64-bit base case. Although, we reduced the total amount of allocated bytes by more than 10%, it seems the case for this benchmark that the ORA technique could only reduce the size of short-lived objects. For

Table 3.1: Number of minor and major GCs under the GenMS and GenCopy collection scheme for the base 64-bit scenario and ORA.

	GenMS				GenCopy			
	minor GCs		major GCs		minor GCs		major GCs	
Benchmark	base	ORA	base	ORA	base	ORA	base	ORA
db	62.7	32.1	2.0	1.0	32.8	59.9	3.0	2.0
jack	184.0	112.7	2.0	1.3	193.9	195.3	2.1	2.1
javac	120.7	117.1	3.5	3.0	115.1	107.5	5.9	5.0
jess	75.9	48.2	0.0	0.0	167.5	81.1	1.0	0.0
antlr	106.7	104.5	5.0	4.3	104.7	89.5	5.9	5.0
fop	25.3	13.3	0.5	0.0	15.2	15.7	1.0	1.0
hsqldb	58.3	49.6	5.0	4.0	60.1	35.6	6.0	5.0
pmd	275.6	302.5	13.0	10.9	214.7	233.5	13.9	12.0
crypt	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
heapsort	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
lufact	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
moldyn	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
search	18.9	18.3	0.0	0.0	21.0	20.2	0.0	0.0
sor	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
sparse	1.0	1.0	0.0	0.0	1.0	1.0	0.0	0.0
pseudojbb	116.9	114.9	4.1	4.0	112.2	109.0	6.2	5.3

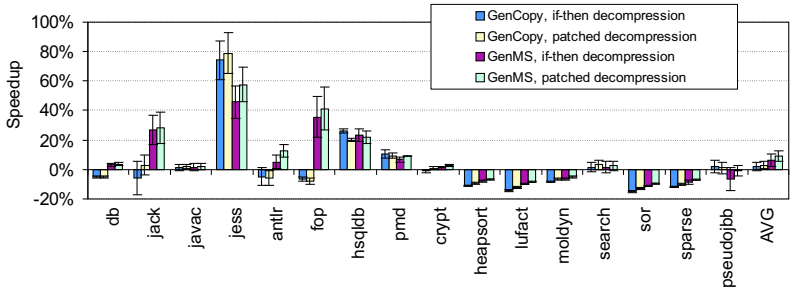

Figure 3.11: Speedup of the garbage collector through ORA.

the javac benchmark, the reduction in maximum reachable bytes (12%) is almost the same as the reduction in total allocated bytes, see Figure 3.7. The reduction in maximum reachable bytes for pseudojbb is somewhere between 5% and 7%. It was expected that this value would be smaller than the total allocated bytes reduction, since we already showed in section 2.4.2 that there is only a limited memory overhead increase from 32-bit to 64-bit mode in terms of maximum reachable bytes.

While Figure 3.9 showed us qualitatively that the number of GCs drops compared to the 64-bit base case, Table 3.1 quantifies the number of GCs performed under ORA. Compared to the 64-bit base case, we observe that the number of GCs drops in correspondence with the reduced number of pages in use as we explained before. On average 10.6% and 17.8% less minor and major GCs occur through ORA, respectively.

Figure 3.11 quantifies the effect of ORA on the total garbage collection time: speedup is shown for a number of ORA configurations and two garbage collectors (GenCopy and GenMS). We observe that for most Java Grande Forum benchmarks, we see small slowdowns, up to 14.9% for sor, while most non- Java Grande Forum benchmarks show a speedup, up to 79.0% for jess. This different behavior of the Java Grande Forum benchmarks is to be expected, since Figure 3.7 already showed us that we could not substantially reduce the memory usage for these applications. On average we observe a small speedup of 5.1%. Although we have a reduced number of garbage collections, the total garbage collection time seems only slightly affected. This can be explained by the extra work that needs to be done by ORA during garbage collection: every traced pointer needs to be decompressed (and compressed again in case of a moving collector), as discussed in section 3.2.6.

3.5 Overall performance evaluation

We now quantify the performance impact of ORA applied to Java application objects. We consider a number of scenarios that we compare with the 64-bit base case. Initially, we assume that all pointer compressions and decompressions occur through the fast path, i.e., all inter-object references can be represented as 32-bit offsets. In section 3.5.2, we then quantify the overhead of patching and the overhead of pointer compression and decompression through the slow path accessing the LAT.

3.5.1 Execution time

Figure 3.12 shows the performance for each of the following four scenarios relative to the base case.

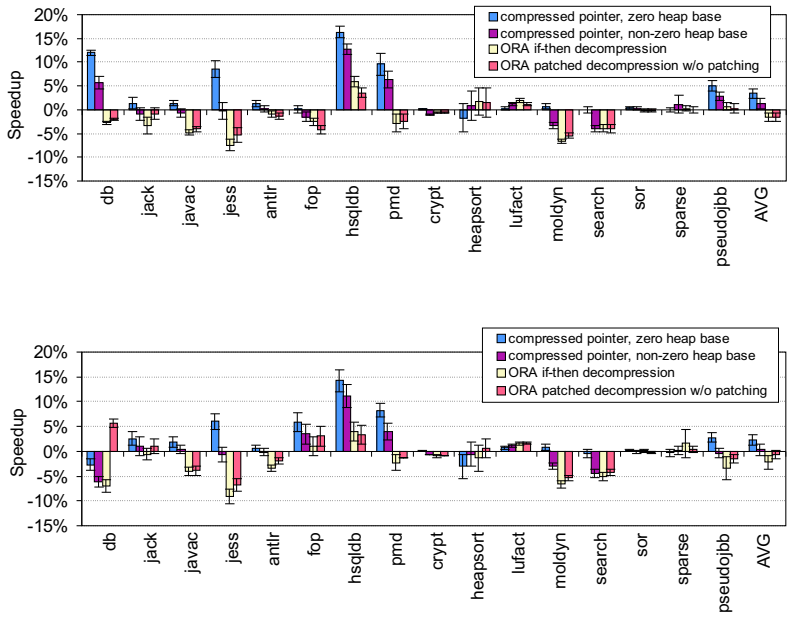

Figure 3.12: Evaluating object-relative addressing in terms of performance: the top graph is for the GenCopy collector, the bottom graph is for the GenMS collector.

Compressed pointers with zero heap base.

The 'compressed pointer with zero heap base' is the scenario where all 64-bit pointers in object data fields are compressed to 32-bit pointers with the heap base address being zero. This means that loading the 32-bit compressed pointers (with zero extension) yields the virtual address of the referenced object; storing a compressed pointer is done by storing the four least significant bytes of the virtual address to memory. This scenario shows the best possible performance that can be achieved through compressed pointer representation: pointers are compressed and there is no compression/decompression overhead. The average performance gain is 2.9%, and up to 16.4% for hsqldb. This performance gain is a direct consequence of the reduced memory usage through a reduced number of data cache misses and D-TLB misses.

Compressed pointers with non-zero heap base.

The 'compressed pointer with non-zero heap base' is similar to the previous scenario except that the heap base address is non-zero. In other words, decompressing a 32-bit pointer requires adding the 32-bit offset to the 64-bit heap base address. This scenario corresponds to Adl-Tabatabai et al.'s approach: it assumes that the heap space is no larger than 4 GB, and assumes a fixed heap base address. The average performance gain for compressed pointers with a non-zero heap base drops to 0.8%; the maximum performance gain is observed for hsqldb (12.7%) and the largest slowdown is observed for db (-6.1%).

The 0.8% average performance gain over the base case is smaller than what is reported by Adl-Tabatabai et al. [3]. The reason is that our results are for the PowerPC ISA using Jikes RVM whereas the results by Adl-Tabatabai et al. are for the Intel Itanium Processor Family (IPF) using ORP and StarJIT. As a result, not all optimizations implemented by Adl-Tabatabai et al. may be implemented in our system. Note however that the goal of this scenario is not to re-validate the approach proposed by Adl-Tabatabai et al., but rather to quantify the overhead of pointer compression/decompression in our framework.

ORA with if-then decompression.

The 'ORA if-then decompression' scenario implements object-relative addressing using the if-then decompression implementation. This scenario includes testing the LSB of the 32-bit compressed pointer for determining whether to take the fast or the slow path. This scenario incurs an average slowdown of 1.9%. The highest slowdown observed is 9.1% (jess); the highest speedup observed is 5.9% (hsqldb).

ORA with patched decompression.

This scenario removes the if-then test from the previous scenario. As explained in section 3.2.2, this version assumes that the slow decompression path is not called for at the beginning of the program execution as the heap is small enough—as such we always take the fast path and thus eliminate executing the if-then statement. Once an inter-object reference is detected that cannot be represented by a 32-bit value, all the code that may possibly read the compressed pointer needs to be patched.

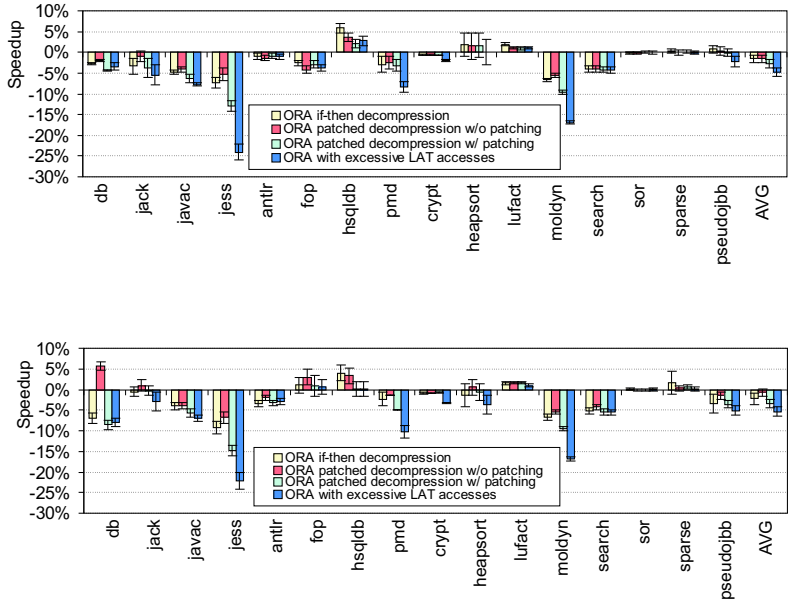

Figure 3.13: Evaluating the overhead of different decompression schemes for object-relative addressing in terms of performance, the top graph for the GenCopy collector, the bottom graph for the GenMS collector.

In this section we will only consider the case where none of the loads are patched, i.e., all pointer decompressions are done by (speculatively) adding the 32-bit offset to the referencing object's virtual address as shown in Figure 3.4. This scenario is labelled 'w/o patching'. Although patching never occurs, the results of this scenario do include all overhead of keeping track of code that potentially needs patching. The overhead of patched code will be discussed in the next section.

This scenario, which eliminates the if-then test in the decompression scheme results in a statistically insignificant average slowdown for the GenMS collector of 0.7% and a small average slowdown of 1.5% for the GenCopy collector. Although this scenario always uses the fast decompression path, we see it as a realistic scenario, since there exist analysis that can guide compilers to prevent too large inter-object references to exist [57].

3.5.2 Overhead evaluation

ORA with patched decompression

In the previous section we only evaluated the patched decompression scenario w/o patching. Now we introduce a second patched decompression scenario, labelled 'w/ patching', which assumes that all loads are patched, i.e., all pointer decompressions are done by jumping to an if-then decompression scheme as shown in Figure 3.5. Figure 3.13 shows the performance of the previous ORA scenarios together with the scenario where all loads are patched. As expected, the 'w/ patching' scenario incurs a higher overhead than the 'if-then decompression' because of the extra jump instruction. The maximum slowdown observed, compared to the 64-bit base case, is 14.8% (jess) and the maximum speedup observed is 2.1% (hsqldb). The measured overhead (on average 3.1%) corresponds to a non-realistic scenario in case all code needs to be patched (the worst case). If patching occurs (again this is a rare event), our ORA implementation will only patch a limited amount of code at once.

LAT access overhead.

So far, we assumed that all decompressions occur along the fast path, i.e., the slow decompression path is never taken. In order to quantify the overhead of going through the slow path we have set up a benchmarking experiment in which the nursery and mature space are located more than 4 GB away from each other. We want to emphasize that the sole purpose of this benchmarking experiment is to quantify the overhead due to taking the slow compression/decompression path; the goal of this experiment is not to present a use case scenario, since for a realistic scenario this would be a bad design choice. This benchmarking experiment implies that all inter-generational pointers — from nursery objects to mature objects, and vice versa — have to pass through the LAT. In other words, a LAT entry is allocated for all inter-generational pointers, and the slow path is taken when compressing/decompressing inter-generational pointers. Although Figure 3.13 also shows the performance of this benchmarking scenario, it is merely illustrative since the overhead depends on the actual number of LAT-accesses. The average slowdown of this benchmarking experiment is 5.0%. On average, 30.9% of all references go through the slow path. By dividing the per-

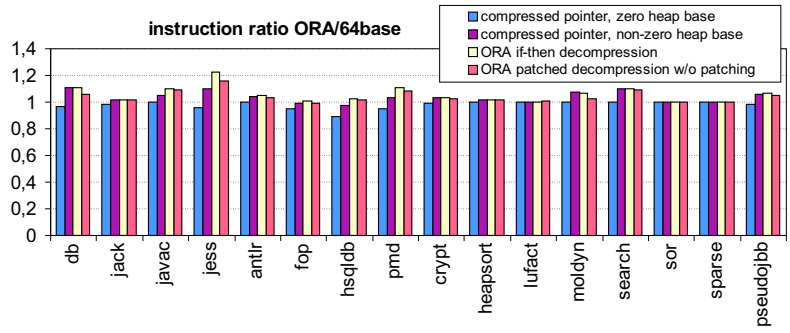

Figure 3.14: Ratio of the number of executed instructions of ORA in relation to the 64-bit base case for the GenMS collector.

formance slowdown by the number of LAT accesses we can compute the performance overhead in terms of cycles per LAT access. We found that the slow compression/decompression path incurs a 5 cycle penalty on average.

3.5.3 Number of instructions executed

Figure 3.14 quantifies the ratio of executed instructions in ORA versus the 64-bit base run. Except for some Java Grande Forum benchmarks, the number of dynamically executed instructions is higher, as expected. ORA has to do extra work when compressing and decompressing pointers. On average, ORA executes 4.3% more instructions. The highest increase (up to 20%) is observed for jess. This is the reason why we saw the biggest performance loss for jess in the previous subsection.

3.5.4 Cache hierarchy performance

Figures 3.15, 3.16 and 3.17 show the number of D-cache misses per 1000 instructions of the base run, for the L1, L2 and L3 level D-cache, respectively. The graphs for the L2 and L3 level are shown twice to magnify the results for benchmarks with a small number of misses per 1000 instructions. In these graphs, we normalize the number of cache misses for the above scenarios to the number of instructions in the base run. We observe that on average the number of L1 misses is only slightly reduced (0.8%). The reduction goes up to 10% for jess. Note that

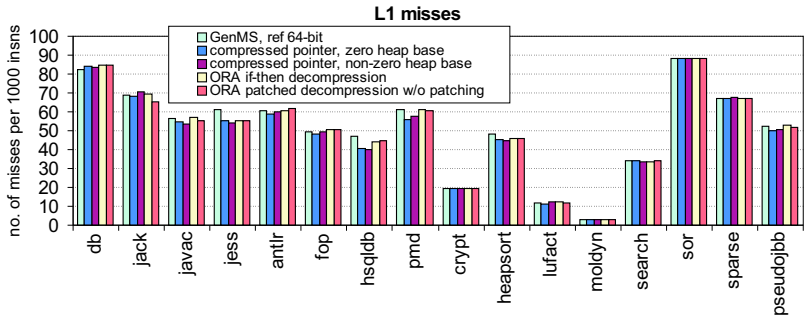

Figure 3.15: The number of L1 D-cache misses per 1000 instructions of the base run for the GenMS collector.

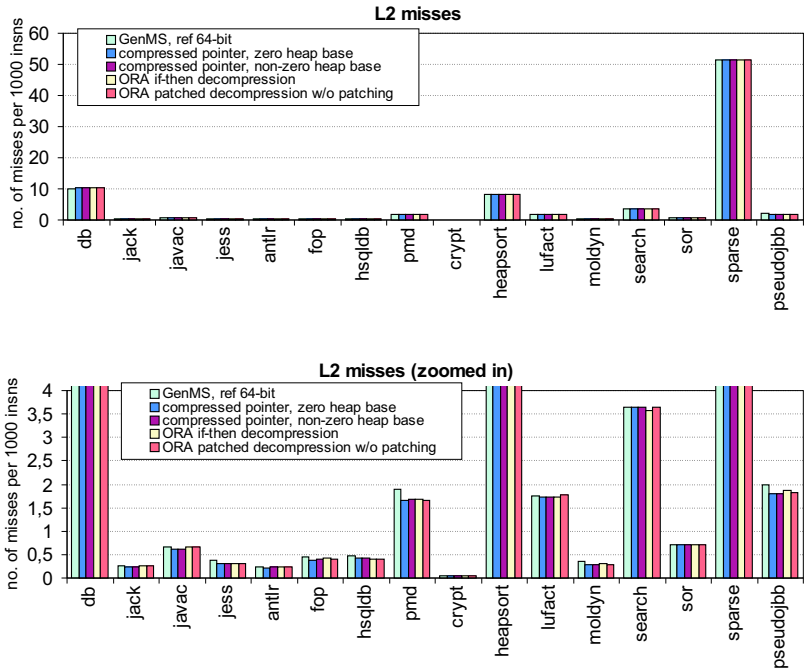

Figure 3.16: The number of L2 D-cache misses per 1000 instructions of the base run for the GenMS collector. The only difference between the first and second graph is the scale of the vertical-axis.

although jess has the largest L1-miss decrease (corresponding to the largest reduction in allocated bytes, see Figure 3.7), jess performs so

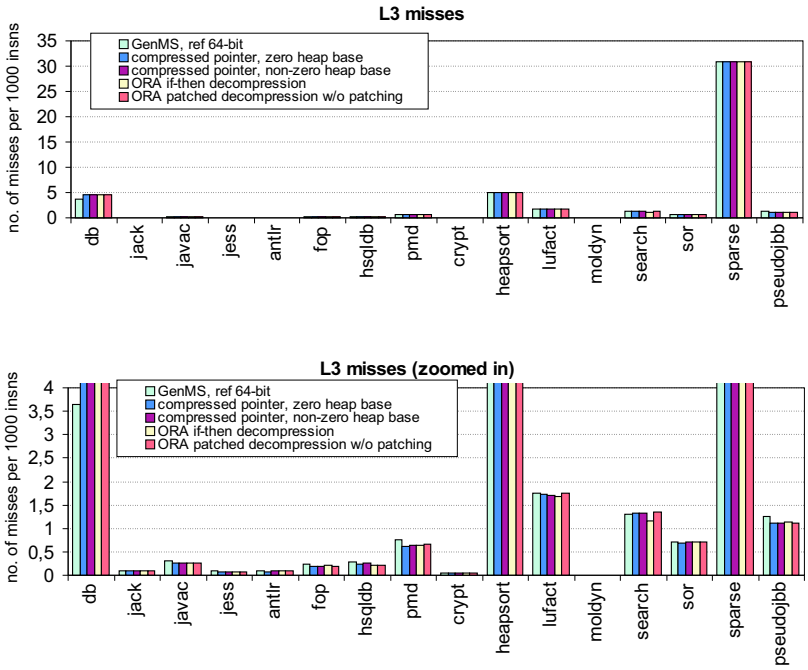

Figure 3.17: The number of L3 D-cache misses per 1000 instructions of the base run for the GenMS collector. The only difference between the first and second graph is the scale of the vertical-axis.

many compressions/decompressions, that the extra cost in terms of performance is much higher than the gain from memory reduction. Only for db, antlr and pseudojbb and only for the GenMS collector, we see a small increase in the number of L1 misses; 2.4%, 2.3% and 1.3%, respectively. Also L2 misses and L3 misses (main memory accesses) are reduced through ORA, on average 5.5% and 6.1%, respectively. The biggest reduction in L2 misses is observed for moldyn (19.6%); observe that moldyn was also the only Java Grande Forum benchmark with a large object size increase in the previous chapter (Table 2.7). With relation to L3 misses, the biggest reduction is seen for hsqldb (27.3%): not surprisingly, also a benchmark with a large object size increase when going from 32-bit mode to 64-bit mode, see Table 2.7. For a single benchmark, namely db, we observe a large increase in L3 misses for the GenMS collector (an increase of 26%). We believe that is due to extra conflict misses that are induced by an accidently bad data layout

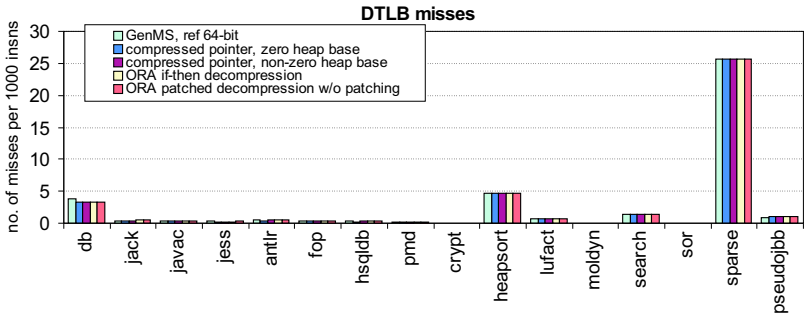

Figure 3.18: The number of D-TLB misses for the GenMS collector, per 1000 instructions of the base run.

on the heap. On overall, we can conclude that ORA better utilizes the cache hierarchy reducing the pressure on main memory.

3.5.5 D-TLB performance

Figure 3.18 shows the number of D-TLB misses per 1000 instructions of the base run for the GenMS collector. The largest reduction (16%) is observed for pmd. One big increase is observed (30%) for fop, but only for the GenCopy collector. On average, ORA reduces the number of D-TLB misses with 4.2%.

3.6 Related work

Adl-Tabatabai et al. [3] address the increased memory requirements of 64-bit Java implementations by compressing 64-bit pointers to 32-bit offsets. They represent 64-bit pointers as 32-bit offsets from a fixed base address of a contiguous memory region. Decompressing a pointer then involves adding the 32-bit offset to a fixed base address yielding a 64-bit virtual address. Compressing a 64-bit pointer results in storing the 32 least significant bits only. The fact that 64-bit virtual addresses are represented as 32-bit offsets from a fixed base address implies that this pointer compression technique is limited to Java programs that demand less than 4 GB of memory.

They apply their pointer compression technique to both the Type Information Block (TIB) pointer — or the vtable pointer — and the for-

warding pointer in the object header and to pointers in the object itself. Separate results for compressing pointers in the object are provided. We do not apply ORA to vtable pointers, because it is highly unlikely that allocating vtables requires more than 4 GB of memory. So, the pointer compression method by Adl-Tabatabai et al. will work properly when applied to vtable pointers. In this chapter, we focus on compressing object references. We will handle object headers (and hence vtable pointers) in the next chapter.

For compressing pointers inside objects only, they report a 13.4 % memory reduction and 14.2 % less GCs. The maximum execution time reduction is observed for `db` (over 50 %). Their memory reduction complies which our measurements. We do not obtain performance results similar to theirs though. There are two reasons for this. First, as mentioned before, the approach by Adl-Tabatabai et al. targets applications that are limited within a 32-bit address space. As such, applications that require more than 4 GB of memory cannot benefit from their pointer compression. ORA does not have that limitation but, in order to achieve that goal, ORA performs a more complicated compression/decompression scheme and hence we can not apply all optimizations applied by Adl-Tabatabai et al., as discussed extensively in section 3.2.7. Second, we use a different experimental setup (hardware platform, virtual machine and memory manager). Measurements in section 3.5.1 for the scenario 'compressed pointer with zero heap base' show at best a 10 % improvement for `db`, while this scenario should perform at least as good as Adl-Tabatabai et al.'s approach.

Lattner and Adve [48, 49] apply a similar approach as Adl-Tabatabai et al. to compressing pointers in linked data structures. Linked data structures are placed in a memory region where pointers are represented relative to the memory region's base address. They need to keep track of different fixed base pointers, one for each region. They can apply their technique only to data structures whose lifetime is bounded by a function's lifetime; ORA on the other hand can apply its compression technique to all pointers. Moreover, ORA implements a safety guard for when certain pointers can not be compressed to 32-bits. Lattner and Adve evaluated their technique for a number of pointer-intensive benchmarks (such as `ks` and `ft` from PtrDist, see also Table 1.2) and one microbenchmark. For these benchmarks, they report a reduction in memory footprint of up to 50% (remark that 50% is the maximum possible reduction, namely the case when all data are pointers). For

some benchmarks they observe almost no memory reduction and even report slowdowns (up to 11%).

Zhang and Gupta [74] compress pairs of 32-bit integer values and 32-bit pointer values to pairs of 15-bit values, each pair packed in a single 32-bit field. Integer values are compressed if the 18 most significant bits are identical. Pointer values are compressed if they share the 17 most significant order bits with their base address, in which case only the 15 least significant order bits are saved. Their scheme has a variable base address for compressing/decompressing pointers, as is the case for ORA. The difference with ORA though is that the offsets in ORA are relative to the referencing object's virtual address. Two addresses can be very close to each other, and hence have different values for their respectively 17 most significant bits (e.g., 0x1234 7F10 and 0x1234 8000). In this case ORA can be applied because they are close to each other, but the technique of Zhang and Gupta can not compress the pointer because the most significant bits differ.

For values that are not compressible by the technique of Zhang and Gupta, extra storage is allocated to store the entire 32-bit values. Since values are always compressed in pairs, this newly allocated storage is 64 bit wide. The value stored at the place of the (pair of) compressed field(s) is then a pointer to the newly allocated storage. The most significant bit is used to indicate whether the *compressed* field contains a packed pair or whether it contains a pointer to unpacked data.

Zhang and Gupta apply their technique to pairs of hot fields and pairs of cold fields in C programs. They evaluated six pointer-intensive benchmarks of the Olden test suite on the simplescalar simulator. They also added extra *data compression extensions* (DCX) to the instruction set. Zhang and Gupta report a 25% allocation space saving and an average reduction in execution time of 12.5% without DCX and 30% with DCX.

Kaehler and Krasner [42] describe the Large Object-Oriented Memory (LOOM) technique for accessing a 32-bit virtual address space on a 16-bit machine. Objects in secondary memory have 32-bit pointers to other objects. Primary (main) memory serves as a cache for secondary memory. Object pointers in main memory are represented as short 16-bit indices into an Object Table. This Object Table contains the full 32-bit address of the object. Objects need to be moved to main memory before they can be referenced. Translation between 32-bit pointers and 16-bit indices is performed when moving objects to main memory.

A number of studies have been done on compressing object headers [8, 63]. They will be discussed more extensively in the next chapter.

Other studies aimed at reducing the memory usage of Java applications, for example, field packing [6], or field reusing [20], or using techniques such as heap compression [21], object compression [19], etc. We will discuss these techniques now briefly.

Ananian and Rinard [6] compute ranges of values that the program may assign to each field. The compiler then transforms the field to the smallest type capable of storing that range of values. They use byte packing to pack different smaller fields in one 32-bit value. They also try to find fields whose values do not change after initialization, or almost always have the same value. They do not store these fields in the object, but provide an alternative access schemes through hash tables.

In [20], Chen et al. examine the lifetime of object fields and they observe that some fields can have disjoint lifetimes. In that case, these fields can share the same memory space.

Chen et al. [21] propose a new memory management strategy that compresses objects in an memory constrained embedded JVM. Their goal is to enable the execution of Java applications using a smaller heap footprint than the original VM can handle. They compress objects as soon as the compaction phase of their GC can not provide enough space for the new object. A compressed object contains a bitmap with one bit for each byte of uncompressed data. If the bit is zero, it means the entire byte is zero, otherwise the non-zero byte is kept in the compressed object. They also apply lazy allocation on portions of large arrays.

In [19], Chen et al. use static analysis to identify fields that frequently have the same value across objects. Fields that are always zero are omitted and other frequent value fields are split off in a separate data structure. These fields in the original object are then replaced with a pointer to this new data structure. These data structures, that contain the frequent fields, are then shared across different objects.

Some other studies investigate hardware compression techniques to compress data in caches [50, 73, 75] or in main memory [29].

3.7 Conclusion

Pointers in 64-bit address spaces require twice as much memory as in 32-bit address spaces. This results in increased memory usage which

degrades cache and TLB performance; in addition, physical memory gets exhausted quicker. This chapter presented object-relative addressing (ORA) for implementation in 64-bit Java virtual machines. ORA compresses 64-bit pointers in object fields into 32-bit offsets relative to the referencing object's virtual address. The important benefit of ORA over prior work, which assumed 32-bit offsets relative to a fixed base address, is that ORA enables pointer compression for programs that allocate more than 4 GB of memory. Our experimental results using Jikes RVM on an IBM POWER4 machine using SPECjvm98, Java Grande Forum, SPECjbb and DaCapo benchmarks show that ORA incurs a statistically insignificant impact on overall performance compared to raw 64-bit pointer representation, while reducing the amount of memory allocated by more than 10% for many benchmarks and up to 16-18% for some benchmarks. The reduction in allocated memory also leads to decreases in the number of D-cache misses: 0.8%, 5.5% and 6.1% on average for the L1, L2 and L3 caches, respectively.

Chapter 4

Selective Typed Virtual Addressing

Nothing fixes a thing so intensely in the memory as the wish to forget it.
Michel de Montaigne

In this chapter we reduce the memory usage of 64-bit Java VM implementations by completely eliminating the object header. This is done through Selective Typed Virtual Addressing (STVA) which means that the object type information is encoded in the object's virtual address. STVA encodes the object type in the object's virtual address by allocating all objects of a given type in a contiguous memory segment. Our results show that STVA yields a reduction in the number of allocated bytes by 15% on average and up to 35% for some benchmarks. Performance is not statistically significantly affected in general, however, some benchmarks exhibit significant overall speedups of up to 20%.

4.1 Introduction

In chapter 1, we pointed out that the increased memory usage is the major drawback for 64-bit systems. Next, chapter 2 identified four reasons why objects in a 64-bit VM occupy more memory than in a 32-bit VM, namely (i) the increased pointer size, (ii) the increased header, (iii) the increased intra-object alignment and (iv) the increased inter-object alignment. The previous chapter, chapter 3, focused on the first reason, and reduced memory usage of 64-bit applications by the object-relative addressing technique. This technique focused on the body part of an

object, also called the object data. In this chapter we will continue working on the topic of object memory reduction, but now we will focus on reasons (ii) and (iv): the header part of an object and the inter-object alignment.

The key technique proposed in this chapter to completely remove the object's header is Typed Virtual Addressing (TVA). By allocating objects of the same type in a contiguous memory segment, type information can be shared by all objects in such a segment and does no longer need to be stored for each object individually. Along with a number of other header layout modifications (which will be detailed further), this allows to remove the Type Information Block (TIB) pointer field as well as the status field from the object header. As such, we are able to completely eliminate the 16-byte object header for non-array objects. For array objects we only keep the 4-byte length field. Accessing the TIB is then done by masking a number of bits from the object's virtual address, and using that as an offset in the TIB space that holds all the TIBs. Removing the status field from the object header is done by keeping GC bits and hash bits in so-called side arrays—1 byte per object in our implementation. Our proposal does not apply TVA to all object types but only to a selected number of types that are frequently allocated, hence the name Selective TVA (STVA). The reason is that applying TVA to all object types results in too much memory fragmentation because of memory pages being sparsely filled with only a few objects.

The idea of typed addressing or implicit typing is not new. Typed addressing has been proposed in the past with proposals such as Big Bag of Pages (BiBOP), typed pointers and others [7, 27, 36, 62, 65]. In fact, it was fairly popular in the 1970s, 1980s and early 1990s in various functional and logic programming languages. However, typed addressing has fallen into disfavor from then on because of the fact that all of these proposals applied typed addressing for all object types. As mentioned above, applying typed addressing to all objects results in memory fragmentation, and eventually performance degradation. With the advent of 64-bit Java implementations, a well designed typed virtual addressing mechanism becomes an interesting option for reducing the memory usage of 64-bit Java VMs because the 64-bit virtual address space is huge which facilitates the implementation of implicit typing compared to 32-bit platforms.

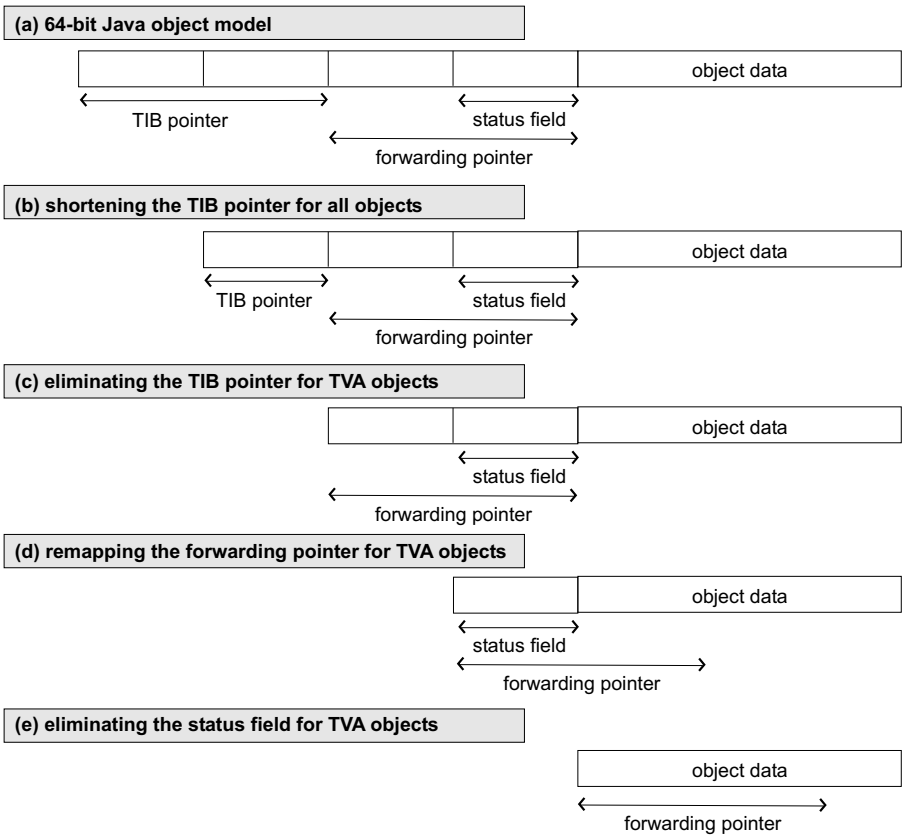

Figure 4.1: The Java (non-array) object models studied in this chapter.

4.2 The 64-bit Java object model

The object model is a key part in the implementation of an object-oriented language and determines how an object is represented in memory. A key property of object-oriented languages is that objects have a run time type. Virtual method calls allow for selecting the appropriate method at run time depending on the run time type of the object. The run time type identifier for an object is typically a pointer to a virtual method table.

An object in an object-oriented language consists of the object data fields along with a header. For clarity, we refer to an object as the object data plus the object header throughout this thesis; the object data refers to the data fields only. The object header contains a number of fields

for bookkeeping purposes. The object header fields and their layout depend on the programming language, the virtual machine, etc. In this work we assume Java objects and we use the Jikes RVM in our experiments. The object model that we present below is for the 64-bit Jikes RVM, however, a similar structure will be observed in other virtual machines, or other object-oriented languages. An object header typically contains the following information, see Figure 4.1(a):

- The first field is the *TIB pointer field*, i.e., a pointer to the Type Information Block (TIB). The TIB holds information that applies to all objects of the same type. In the Jikes RVM, the TIB is a structure that contains the virtual method table, i.e., a pointer to an object that represents the object type and a number of other pointers for facilitating interface invocation and dynamic type checking. The TIB pointer is 8 bytes in size on a 64-bit platform.
- The second field is the *status field*. The status field can be further detailed into a number of elements.
  - The first element in the status field is the *hash code*. Each Java object has a hash code that remains constant throughout the program execution. Depending on the chosen implementation in the Jikes RVM, the hash code can be a 10-bit hash field in the header or a 2-bit hash state.
  - The second element in the status field is the *lock element* which determines whether the object is being locked. All objects contain such a lock element. A thin lock field [9] in the Jikes RVM is 20 bits in size.
  - The third element is related to *garbage collection*. This could be a single bit that is used for marking the object during a mark-and-sweep garbage collection. Or this could be a number of bits (typically two) for a copying or reference counting garbage collector.
- The third field is the *forwarding pointer*. The forwarding pointer is used for keeping track of objects during generational or copying garbage collection and is 8 bytes in size. The forwarding pointer overwrites the hash code and lock element in the status field, but not the garbage collection bits. The garbage collection bits are chosen as the least significant bits so that they do not get overwritten by the forwarding pointer (due to 8-byte alignment of the forwarding pointer).

So far, we considered non-array objects. For array objects there is an additional 4-byte length field that needs to be added to the object header. As a result, for array objects the header field requires at least 20 bytes. But given the fact that alignment usually requires objects to start on 8-byte boundaries on a 64-bit platform, the array object header typically uses 24 bytes of storage.

4.3 Eliminating the header in the 64-bit Java object model

We eliminate the header of 64-bit Java objects in a number of steps. Our initial Java object model is the 16-byte header as shown in Figure 4.1(a) — we limit the discussion to non-array objects for now, and will discuss array objects later.

- We first reduce the TIB pointer size from 64-bit to 32-bit through pointer compression as proposed by [3]. This is shown in Figure 4.1(b). This object model implies that all the TIBs are allocated in a contiguous virtual address space that is small enough to be accessed using a 32-bit offset. The TIB pointer is then computed by adding the 32-bit TIB pointer stored in the object header to a 64-bit TIB base pointer. TIB pointer compression is applied to all objects.

- As a second step we apply Selective Typed Virtual Addressing (STVA) to completely eliminate the TIB pointer from the object header, see Figure 4.1(c). STVA applies Typed Virtual Addressing (TVA) to a selected number of object types.

- In the third step, we map the forwarding pointer in a different way so that the forwarding pointer overlaps with the 4-byte status field and the first four bytes of the object data, see Figure 4.1(d). Note that the object data is already copied during garbage collection whenever the forwarding pointer gets used. This allows us to freely overwrite the first four bytes of the object data[1]. This layout requires to move the GC status bits from the least significant status field bit positions to the most significant

[1]Obviously, this cannot be done for data structures belonging to the garbage collector itself—this is only of concern whenever the garbage collector (or the entire VM) is written in Java.

status field bit positions so that the GC bits are not overwritten by the forwarding pointer.

- As a final step, we remove the status field from the object header, see Figure 4.1(e). As discussed in section 4.2, the status field contains multiple components. The removal of each of these components will be explained in detail in section 4.5.1. The end result is that the object header is completely eliminated for all TVA-enabled objects.

In the following sections, we discuss STVA in detail since it is the key enabler for completely eliminating the object header.

4.4 TIB pointer compression

In addition to removing the header for all TVA-enabled objects, we also compress the 64-bit TIB pointers to 32-bit pointers for all TVA-disabled objects. Previous work [3] proposed pointer compression to all pointers (not just TIB pointers) in a 64-bit VM implementation. However, the limitation of compressing all pointers is that applications that require more than a 32-bit virtual address space cannot benefit from this pointer compression approach. Applying pointer compression to TIB pointers only does not suffer from this limitation; the case where more than a 32-bit virtual address space is needed for type information only is highly unlikely. Pointer compression inside the object data was studied in more detail in chapter 3. Here we will only consider the object header.

4.5 Selective Typed Virtual Addressing

This section explains the idea and the implementation details behind Selective Typed Virtual Addressing (STVA) for 64-bit Java objects. We first detail on the TVA 64-bit Java non-array object model followed by the TVA array object model. We then go through a number of virtual machine implementation issues that follow from the TVA object models.

4.5.1 The non-array TVA object model

We consider two TVA Java object models: the small-header object model and the no-header object model.

The small-header object model

The small-header TVA object model eliminates the TIB pointer from the object header and remaps the forwarding pointer to overwrite the 4-byte status field and the first 4 bytes of the object data, see Figure 4.1(d). This implies that the minimum size occupied by a TVA-enabled object is 8 bytes: 4 bytes of object header and 4 bytes of data. We will refer to the obtained object model in Figure 4.1(d) as the small-header object model throughout this chapter.

The no-header object model

The no-header object model extends on the small-header object model by eliminating the 4-byte status field, see Figure 4.1(e). This requires that we eliminate the lock, the hash and the GC elements, as well as the forwarding pointer from the object header. This is done as follows.

- We eliminate the GC elements from the object header by storing the GC elements in what we call a *side array*. We implement a side array every two pages on which TVA-enabled objects are allocated and allocate one byte per object in the side array. Note that depending on the garbage collector only one or two bits are used from the allocated byte in the side array. The position in the side array for a given object is determined by the position of the object on the memory page. Note that this can be done because all objects in a TVA region are of the same type and thus are equally sized. By consequence, during garbage collection, we do not adjust the GC elements in the header for TVA-enabled objects but we adjust the GC elements in the side arrays. The index in the side array is computed from the object's virtual address which incurs an additional overhead compared to the default and small-header object models.
- Dealing with the lock element in the no-header format is done along what is described in [8]. Objects from a class that contains at least one `synchronized` method (or at least one of the class

methods contains the `synchronized(this)` statement) have an additional implicit field member that contains the lock. This is a 4-byte field in our implementation. This implicit lock field scheme cannot be applied to arrays or in cases where a lock is taken on the general `object` type. In these cases or in case a lock needs to be taken on a different object type, a new lock object is then created in the lock nursery [8].

- The hash elements in the Java object model can take three states: *unhashed*, *hashed* and *hashed-and-moved* [4, 8]. For the first two states, unhashed and hashed, the hash code is calculated from the object's address. In case the garbage collector moves an object that is in the *hashed* state, its state then changes to *hashed-and-moved* and the hash code is attached to the end of the new version of the object. In the traditional Java object model as well as in the small-header object model, these three states are encoded using two hash bits. In the no-header object model on the other hand, we store only one bit per TVA-enabled object in the side arrays just described. The hash bit in the side array is zero if the object is *unhashed*; the hash bit in the side array is set if the object is *hashed*. When a *hashed* object is moved by the garbage collector, we TVA-disable the object, i.e., the object moves from the TVA space to the non-TVA space.

- The forwarding pointer, whenever needed during garbage collection, overwrites the first 8 bytes of the object data. This implies that the minimum size for a TVA-enabled object is 8 bytes — this is the same as for the small-header object model.

The advantage of using a side array over a small header is twofold. First, a side array consumes less memory, since a header needs to be minimal 4 bytes long due to alignment issues. Second, accesses to the meta data stored in the header or side array and accesses to the actual object data are often uncorrelated. In particular, during garbage collection we often only require the meta data, during program execution mostly object data is accessed. So by separating these two parts physically in memory, locality of either parts might improve. The disadvantage of a side array over a small header, is that it takes an extra computation to access the side array.

4.5.2 The array TVA object model

The array TVA object model more or less follows the same lines as the non-array TVA object model, however, there are some peculiarities in relation to memory management. In case of a copying collector, the memory allocator typically uses a bump pointer to allocate new objects, i.e., the bump pointer is incremented by the size of the newly allocated object. In case of a mark-sweep collector, at least in the Jikes RVM, the memory allocator works with fixed-sized cells. The choice of the memory management method affects the array TVA object model.

The small-header object model

The small-header TVA array object model has an 8-byte header consisting of a 4-byte status field and a 4-byte array length field. We can select all arrays of all lengths to be TVA-enabled for a copying collector. Although selecting all arrays is also possible in case of a mark-sweep collector, this would result in a considerable run time overhead and memory fragmentation because of the fixed-sized cells in the Jikes RVM implementation. Therefore, in case of a mark-sweep collector, we only select a single array length to be TVA-enabled[2].

The no-header object model

The no-header TVA array object model eliminates the same header fields as the no-header TVA non-array object model, hence only the 4-byte length remains in the header. In order not to incur a large run time overhead, we select at most one array length on which to apply TVA for both the copying and the mark-sweep collectors. The underlying reason is that a single array length eases accessing the side arrays; the side array index can be computed directly from the object's address.

4.5.3 Implications of the TVA object model

We now focus on a number of implications because of the TVA Java object model. Although some of these issues are geared towards our implementation in the Jikes RVM on an IBM AIX system, similar issues will need to be taken care of on other systems.

[2]We only selected a single array length, although it is possible to experiment with a small number of array lengths as well.

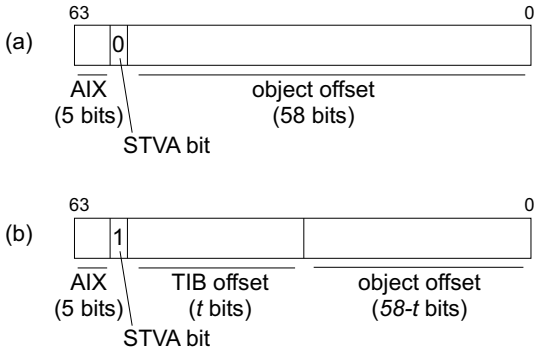

Figure 4.2: The 64-bit virtual address for a TVA-disabled object (a) and for a TVA-enabled object (b).

Memory allocation

The general idea behind Typed Virtual Addressing is to devote segments (large contiguous chunks of memory) in the virtual address space to specific object types. This means that the object type is implicitly encoded in the object's virtual address. Object types that fall under TVA are then allocated in particular segments of the virtual address space. For example, all objects of type A get allocated in the virtual address space segment with addresses ranging from address `0x04FF FFFE 0000 0000` to address `0x04FF FFFF FFFF FFFF`; all objects of type B then get allocated in the virtual address space segment in the range `0x0500 0000 0000 0000` to `0x0500 0003 FFFF FFFF`.

The virtual memory address of a Java object in an STVA-enabled VM implementation is depicted in Figure 4.2. The five most significant bits are AIX-reserved bits and should be set to zero. The following bit (bit 58) is the STVA bit that determines whether the given object falls under TVA. This divides the virtual address space in two regions, the TVA-disabled region and the TVA-enabled region. Note that although we consume half of the virtual address range for TVA-enabled object types, we leave 2^{58} bytes for TVA-disabled object types. If bit 58 is set, then the object is a TVA-enabled object, i.e., the object follows the TVA Java object model detailed in section 4.5.1. If bit 58 is not set (the object type is a TVA-disabled type), the object falls under the default Java object model from Figure 4.1(a). In the latter case (a TVA-disabled object type), the least significant 58 bits determine the object's offset, see Figure 4.2(a). In case of a TVA-enabled object, see Figure 4.2(b), the next

t bits of the virtual address constitute the TIB offset (t equals 25 in our implementation). The TIB offset determines in what memory segment the objects of the given type reside. By doing so, an object type specific memory segment is a contiguous chunk of memory of size 2^{58-t} bytes; this is 8 GB in our implementation. The least $(58-t)$ significant bits are the object offset bits (33 bits in our implementation). These bits indicate the object's offset within its type specific segment.

In order to support this memory layout, we obviously need to modify the memory allocator to support TVA. We now need to keep track of multiple allocation pointers that point to the free space in the object type specific segments in order to know where to allocate the next object of the given object type. The selection of an individual allocation pointer requires an extra indirection for TVA-enabled object types. We eliminate this additional indirection by refactoring the code, i.e., by inlining the allocation pointer array.

Another peculiarity related to the no-header TVA memory allocators is that we know that all objects within a type-specific segment have equal sizes. With this knowledge we can layout fixed sized cells into the TVA-enabled regions, prior to allocation. This layout will include proper alignment, so that we are able to remove the alignment burden from the memory allocator. If it is known that the specific object type does not require 8-byte alignment (i.e., no references in its data fields), this type can be pre-aligned on a 4-byte multiple in the TVA-enabled region. Hence this pre-alignment potentially reduces the extra inter-object alignment overhead on 64-bit versus 32-bit mode (reason (iv) in this chapter's introduction).

Yet another peculiarity relates to copying collectors. A traditional copying collector needs to figure out the size of the object to be allocated. This is done by accessing the TIB, reading the pointer that points to the object that represents its class, and retrieving the object size from the class object. In our TVA-aware copying collector, we keep track of the object sizes for the various object types in an array structure. Thus, a single array lookup yields us the object size to be allocated.

TIB access

In an STVA-aware VM implementation, reading the TIB pointer changes compared to a traditional VM implementation. In a traditional implementation (without STVA), the TIB pointer is read from the object

```
    ;; R3 contains the object's virtual address

    tst R3, 0x0400 0000 0000 0000  ;; test bit 58
    bre L2               ;; jump to L3 in case bit is not set

L1: ;; TVA-enabled object:
    ;; mask the TIB offset from the object's virtual address
    rsh R4, R3, (64 - FIXED_BITS - NUM_TIB_BITS)

    lsh R4, R4, 3        ;; align offset to 8 bytes
    add R4, TIB_BASE, R4  ;; add TIB offset to the base TIB
                  ;; pointer; the constant TIB_BASE equals
                  ;; the real base TIB pointer with bit 58
                  ;; set to zero -- as an optimization, bit
                  ;; 58 is not masked away.
    jmp L3

L2: ;;TVA-disabled object: load TIB pointer from the header
    ld R4, R3, TIB_OFFSET

L3: ...                  ;; R4 contains the TIB value
```

Figure 4.3: Computing an object's TIB pointer in an STVA-enabled VM implementation.

header through a load instruction. In an STVA-aware VM implementation, we make a distinction between a TVA-enabled object and a TVA-disabled object. This is illustrated in pseudo-code in Figure 4.3. A TVA-disabled object follows the traditional way of getting to the TIB pointer. A load instruction reads the TIB pointer from the object header. For a TVA-enabled object, the TIB pointer is computed from the object's virtual address. This is done by masking the TIB offset from the virtual address and by adding this TIB offset to the TIB base pointer — all the TIBs from all object types are allocated in a limited address space starting at the TIB base pointer. The size of the TIB space is limited to 256 MB in our implementation; this comes from the 25-bit TIB offset that we use, see Figure 4.2, along with a 3-bit shift left for 8-byte alignment. Note again that this is not a hard limit and can be easily adjusted by changing the address organization from Figure 4.2 in case a 256 MB TIB space would be too small for a given application (which is unlikely for contemporary applications).

Due to the conditional jump for determining the TIB pointer, see Figure 4.3, our STVA-enabled implementation clearly has an overhead

compared to a traditional VM implementation. The single most impediment to implementing STVA more efficiently is the branch that is conditionally dependent on whether the object is TVA-enabled or TVA-disabled. Unfortunately, in our PowerPC implementation we were not able to remove this conditional branch through predication. Nevertheless, this could be a viable solution on ISAs that support predication, for example through the `cmov` instruction in the Alpha ISA, or through full predication in the Itanium ISA.

As an optimization to computing the TIB pointer, we limit the frequency of going through the relatively slow TIB access path.[3] This is done by marking the class tree with the TVA-enabled object types. A subtree is marked in case all types in this subtree are TVA-disabled. The TIB access then follows the fast TIB access path as in a non STVA-aware VM.

Since the TIB offset is computed from an object's virtual address, the position in memory of the TIB is obviously related to the object type specific memory segment. We cannot position the TIB independently from the object type specific memory segment. To address this problem, we make sure we first allocate the TIB in the TIB space. This will give us the TIB offset to be used for all objects of the given TVA object type. Once the type specific memory segment for a TVA object type is properly initialized, TVA-enabled objects can be allocated in it.

Impact on garbage collection

Implementing TVA obviously also has an impact on garbage collection. In this section we discuss garbage collection issues under the assumption of a generational garbage collector which is a widely used garbage collector type. Similar issues will apply to other collectors as well. In a generational collector, there are two generations, the nursery and the mature generation. Objects first get allocated in the nursery. When the nursery fills up, a nursery collection is triggered and reachable objects are copied to the mature generation. New objects then get allocated from an empty nursery. This goes on until also the mature generation fills up. When the mature generation is full, a major heap collection is triggered.

In the original Jikes RVM implementation with a generational col-

[3] This is only possible in the offline STVA implementation, see later for a discussion on the offline STVA implementation.

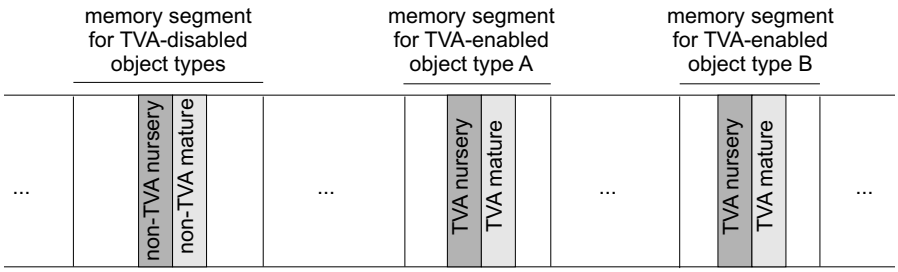

Figure 4.4: Mapping the nursery and mature spaces in the virtual address space in a TVA-aware VM.

lector, the nursery and mature generations consist of contiguous spaces. This means that there is one or two contiguous spaces for the nursery and mature generations. In an STVA-aware VM implementation, contiguous memory segments are defined for specific object types that fall under TVA, but the union of all these memory segments is no longer contiguous. Because the nursery and mature spaces need to fall within all type-specific memory segments, these spaces can obviously no longer be contiguous. As such, we end up with non-contiguous spaces in both the nursery and mature generations. The nursery generation now consists of a contiguous space for TVA-disabled object types, and a non-contiguous space for TVA-enabled object types. The mature generation is constructed in a similar way. This is illustrated in Figure 4.4.

Jikes RVM however, works with contiguous spaces. Jikes RVM identifies a space by a SpaceDescriptor, a numerical value encoding the nature, size and starting address of the space. In order to be able to use non-contiguous spaces in our TVA-aware VM, we extended Jikes RVM's implementation of a space. In our system, we identify a space by the combination of its starting address and its mask. If we represent a space i by S_i, its mask by M_i, and its starting address as B_i, and if we represent an address by A, then the following is true by definition:

$$B_i \& M_i = B_i$$
$$A \in S_i \Leftrightarrow A \& M_i = B_i$$

with $\&$ being the bitwise 'and' operator. A mask M consists of one or more series of `'1'`s and one or more series of `'0'`s; the following bit patterns are examples:

$$`00..011..100..0', `11..100..0'$$

A contiguous space is just a special case in which the mask has a leading series of `'1'`s followed by a trailing series of `'0'`s, i.e., the mask looks as follows:

$$\text{`}\texttt{11..100..0}'$$

A non-contiguous space has a mask of any other form that does not consist of a leading series of `'1'`s followed by a trailing series of `'0'`s, for example:

$$\text{`}\texttt{00..011..100..0}'$$
$$\text{`}\texttt{11..100..011.100..0}'$$
$$\text{`}\texttt{00..011..100..011..100..0}'$$

We also have a simple rule to check if two spaces are non-overlapping. This is needed when allocating spaces:

$$\neg(A \in S_i \wedge A \in S_j) \Leftrightarrow (M_i \& M_j \neq 0) \wedge ((B_j \& M_i) \neq (B_i \& M_j)), \forall i, j$$

Note that the mask is part of the definition of a space. As such, each space has a dedicated mask that does not need to be computed at run time; however, the mask is part of the TVA-aware VM implementation.

4.6 STVA type selection

As mentioned before, we do not apply TVA to all objects. Object types that are allocated infrequently would occupy memory pages that are only sparsely filled with objects. This would result in too much memory fragmentation. To limit memory fragmentation we need to limit the number of object types on to which TVA is applied. We believe that this is a key difference to prior work on typed virtual memory addressing. Prior work applied TVA to all object types. We propose to limit TVA to only a subset of well chosen object types in order to control the amount of memory fragmentation while pertaining the benefits of typed virtual addressing. We now explore two approaches to selecting object types on which to apply TVA, namely an offline selection strategy and an online selection strategy.

4.6.1 Offline STVA type selection

In our offline STVA implementation, we apply the following strategy for making an object type TVA-enabled. In order to select an object type to fall under TVA, the object type needs to apply to one of the following

two criteria. First, an object type needs to be allocated frequently, and second, its instances are preferably long-lived. In our first criterion we make a selection of object types of which a sufficient amount of objects is allocated. Through a profiling run of the application, we collect how many object allocations are done for each object type, and what the object size is for each object type. Once this information is collected, we compute for each type the total number of allocated header bytes (16 bytes per instance), and we compute the percentage volume of these header bytes in relation to the total number of allocated bytes. We then select object types for which this percentage volume exceeds a given *memory reduction threshold (MRT)*.

In our second criterion we limit the scope to long-lived objects because long-lived objects are likely to survive garbage collections. These objects will thus remain on the heap for a fairly long period of time. Giving preference to long-lived objects under TVA maximizes the potential memory savings. In order to classify object types into long-lived and short-lived object types, we take a pragmatic approach and inspect a profile run of the application for objects that survive garbage collections. In these runs we use a fairly large heap in order to identify truly long-lived objects. For those objects that survive a garbage collection, we again compute the percentage volume of the header bytes in relation to the total number of bytes surviving the collection. We then retain object types for which this percentage volume exceeds the *long-lived memory reduction threshold (LLMRT)*.

4.6.2 Online STVA type selection

An important disadvantage of the offline STVA type selection method is that a profiling run is needed for determining on what object types to apply TVA. This is not practical in the context of a Virtual Execution Environment. Therefore we now propose on online STVA type selection mechanism. The mechanism that we propose is a simple but effective approach. When re-compiling a method in the VM, we TVA-enable all the object types that are allocated within these re-compiled methods. The underlying idea is that frequently executed methods, so called hot methods, are scheduled for re-compilation and re-optimization; if these methods allocate objects, they will allocate lots of these objects. In other words, the types of the objects allocated in methods that are scheduled for optimization, are likely to be frequently allocated object types. By consequence, these object types are good candidates for STVA selection.

Note that the online STVA type selection method is different from the offline approach. The reason is that a direct translation of the offline approach into an online approach would be fairly complex. This translation would require that we keep track of the number of bytes allocated for each type at run time. In addition, we have to determine *when* to convert an object type from TVA-disabled to TVA-enabled. And once we have determined when to convert an object type, we then have to recompile the methods allocating this object type. This is a fairly complex mechanism. Instead we have chosen for a much simpler alternative that triggers TVA for objects allocated in recompiled methods; this is motivated by the fact that methods need to be recompiled anyway in order to enable TVA for objects allocated in these methods. So, as for the *when* part, we choose recompilation time[4] for triggering TVA, and for the *which object types* part, instead of determining frequently used object types, we simply select all objects allocated in hot methods. This simple approach showed to perform well in practice, as we will demonstrate in the evaluation section of this chapter.

In case of the no-header object model, we do not select array types online because it is difficult for an online mechanism to select what array length to support under TVA. Also, in order to limit the total number of STVA types, we limit the number of TVA-enabled object types to a maximum which is 85 types in our implementation (which is about the maximum observed through our offline STVA type selection method even under the lowest MRT thresholds, as shown in the evaluation section). For the benchmarks that we ran, only javac and pmd ran against this limit.

Note that some object types are selected when building the TVA-aware VM. These are the so called *bootlist* TVA-enabled object types. In other words, all object types from this bootlist are TVA-enabled for all applications running on this VM. The bootlist TVA-enabled object types were chosen out of the object types as selected through the offline STVA type selection method from the previous subsection. The bootlist contains these (VM and library) object types that are frequently allocated across various application runs.

[4]The optimizing compiler in Jikes RVM uses sampling to detect recompilation candidates. In all of our measurements, we let the adaptive system of Jikes take the recompilation decisions, without enforcing certain methods being opt-recompiled. We acknowledge that there might be small fluctuations in selected methods between different runs.

4.7 Experimental setup

Our experimental setup is the same as for the previous chapters. We use the Jikes RVM version 2.3.5, with the GenCopy and GenMS garbage collectors[5]. The hardware platform on which we have done our experiments is the IBM POWER4. The benchmarks are taken from a variety of suites (SPECjvm98, SPECjbb2000, DaCapo and the Java Grande Forum benchmarks). In order to be able to draw statistically valid conclusions, we employ statistics to determine 95% confidence intervals from 15 measurement runs. These statistics will help us in determining whether the reduced header object models result in statistically significant or statistically insignificant performance gains or degradations. The experimental setup is described in detail in section 2.2.

4.8 Evaluation

We now evaluate the reduced header Java object models using the experimental setup detailed in the previous section.

4.8.1 Feasibility study of STVA

We first inspect the potential of Selective Typed Virtual Addressing. Since for small objects, the object header (fixed size for all objects) occupies a big part of the total object size, we already have an indication that removing the object header from these objects has a big potential. Indeed, in chapter 2 we already quantified the increase in object size, of the transition from 32-bit mode to 64-bit mode, for small objects. On average, the object's size increases with about 20 bytes (see Table 2.8), of which 16 comes from the header increase. So applying STVA to these (small) objects, could decrease their size even further than the 32-bit object size. In order to verify and quantify this potential, we now characterize the profile input. As mentioned in section 4.6, we determine whether an object type is TVA-enabled or TVA-disabled based on two criteria for offline STVA type selection. First, a type needs to be allocated frequently, i.e., the potential memory savings for the given type need to exceed the memory reduction threshold (MRT). Or, second,

[5]The STVA technique can also be implemented for all other garbage collection schemes. We limit our implementation to the two best performing garbage collection algorithms available in Jikes RVM 2.3.5.

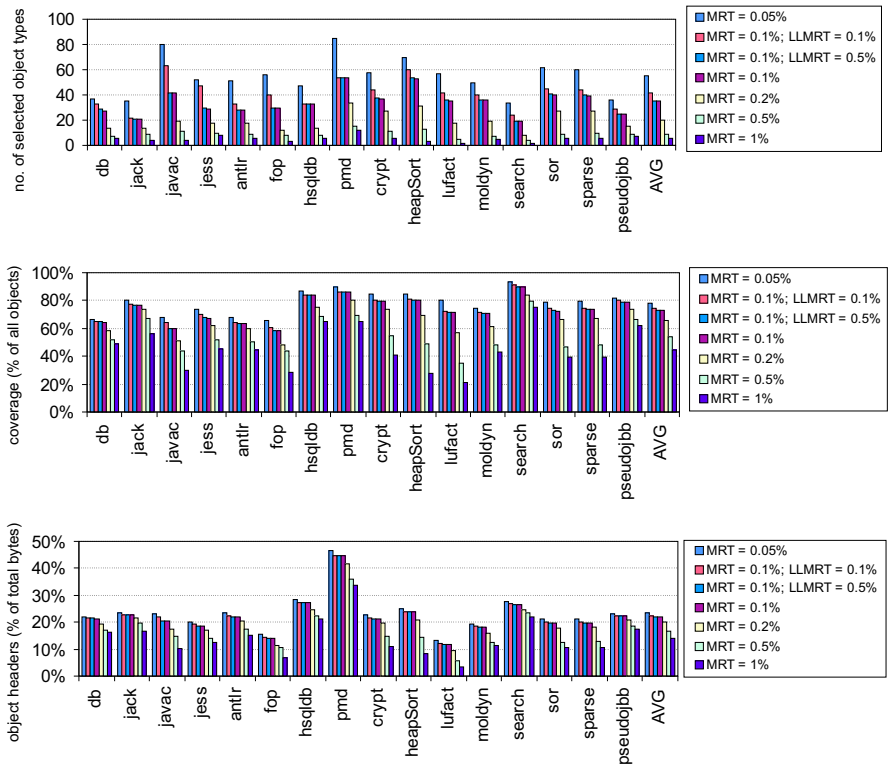

Figure 4.5: The top graph shows the number of selected object types as a function of the MRT and LLMRT thresholds. The middle graph shows the coverage by the selected objects as a percentage of the total number of objects. The bottom graph shows the number of allocated bytes in the headers of the selected object types as a percentage of the total number of allocated bytes.

the potential memory savings for the given type in case it is a long-lived type need to exceed the long-lived memory reduction threshold (LLMRT). We now study the sensitivity of the number of selected object types and the potential memory savings to the chosen MRT and LLMRT thresholds. This is shown in Figure 4.5 by varying MRT from 0.05% up to 1% and by varying LLMRT over three values 0.1%, 0.5% and infinite. Note that the data in Figure 4.5 is for the profile input, and only gives a rough indication of what is to be expected for the reference input. In addition, Figure 4.5 only contains data concerning the nursery and mature generations. The data allocated in the Large Object Space (LOS) — the LOS is the space in which all large objects get allocated —

is removed from this graph for clarity; STVA is expected to give only a very marginal benefit for large objects.

The top graph in Figure 4.5 shows the number of selected object types. As expected, we observe that the number of selected types decreases with increasing MRT and LLMRT. For example, an MRT of 0.05% selects on average 55.6 types whereas an MRT of 0.2% selects on average 20 types. The number of selected types varies across the benchmarks; for example for a 0.2% MRT, the number of selected objects varies from 8 up to 34. Note that this is only a small fraction of the total number of object types. The total number of types allocated at least once ranges from 450 to 2800 across the various benchmarks. The middle graph in Figure 4.5 shows the coverage by the selected object types, i.e., the fraction of the total number of allocated objects that is accounted for by the selected object types. We observe that selecting only a small number of types results in a fairly large coverage. A 0.05% MRT yields an average coverage of 78.3%; a 0.2% MRT yields an average coverage of 65.8%. The bottom graph in Figure 4.5 shows the percentage of the total number of allocated bytes due to headers of the selected object types. This percentage shows the potential memory savings in case the complete header would be removed from the selected objects for the profile input. For example, a 0.05% MRT potentially yields an average 23.5% potential reduction in allocated bytes, with a peak for pmd of 46.7%. A 0.2% MRT yields an average potential reduction of 20%.

4.8.2 Memory usage and impact on GC

Figures 4.6 and 4.7 show the reduction in allocated bytes for the offline and online header reduction techniques, respectively. For the offline technique, these numbers are for the reference input from a cross-validation setup. We observe an average reduction in allocated bytes of 15%. For some benchmarks we even observe a reduction in allocated bytes of 26% (antlr), 28% (search) and 35% (db). There are a number of important notes that we would like to make:

- Our first note relates to the data presented in Figure 4.6 compared to the data presented in Figure 4.5 for the feasibility study. The data in Figure 4.6 is for the reference runs whereas Figure 4.5 is for the profile input. Note that for some benchmarks such as db we obtain larger memory savings with the reference input than what we expected from the profile input, compare Figure 4.6 with

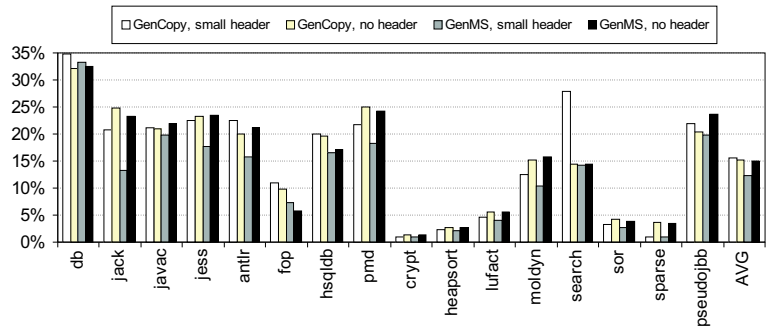

Figure 4.6: Reduction in the number of allocated bytes for the offline header reduction techniques with MRT and LLMRT set to 0.1%.

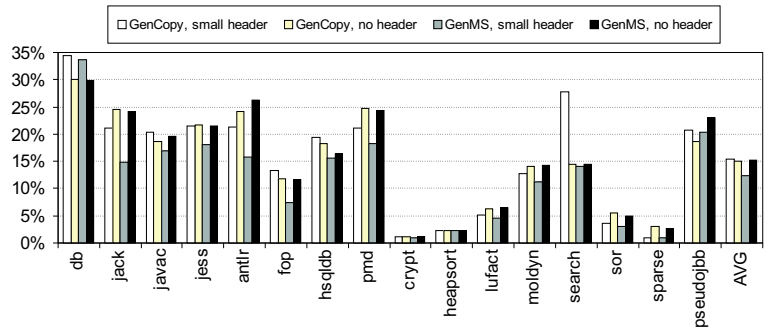

Figure 4.7: Reduction in the number of allocated bytes for the online header reduction techniques.

Figure 4.5. This is explained by the fact that the reference input spends more time in the application than the profile input does. And since the VM objects tend to be larger than application objects, it is to be understood that the memory savings are larger for the reference input than for the profile input.

- A second note we would like to make is that some benchmarks, such as some of the JGF benchmarks, have a fairly low reduction in allocated bytes. The reason is that these benchmarks allocate long arrays — reducing the header size thus has a limited effect on the overall memory reduction. In addition, the data in Figure 4.5 shows potential memory reductions in the nursery and mature generations only, no data is included concerning the large

benchmark	offline	online	in common
db	35	38	35
jack	37	43	36
javac	56	85	48
jess	41	44	38
antlr	41	60	38
fop	41	35	33
hsqldb	36	53	36
pmd	59	85	42
crypt	33	34	32
heapSort	34	33	32
lufact	34	32	32
moldyn	35	33	32
search	34	33	32
sor	33	32	32
sparse	33	33	32
pseudojbb	44	57	39

Table 4.1: Number of TVA-enabled object types for offline STVA type selection, online STVA type selection and the number of object types in common between offline and online type selection. This includes 32 bootlist TVA-enabled object types.

object space (LOS). The data in Figures 4.6 and 4.7 show the effective memory reduction.

- A third note is that for some benchmarks, the no-header object model allocates more bytes than the small-header object model. There are two reasons for this. First, in case of a copying collector, the small-header object model is applied to arrays of all lengths whereas the no-header object model is only applied to arrays of a single length as discussed in section 4.5.2. Some benchmarks suffer from the fact that TVA cannot be applied to all array sizes. Second, when a TVA-enabled object, on which a hashcode is taken, is moved in the no-header object model, the object is TVA-disabled which causes the object to grow in size.

- Finally, the reduction in allocated bytes is comparable between the offline and online header reduction techniques, in spite of the different approaches taken for selecting TVA-enabled object types, as discussed in section 4.6. The reason is that the offline and online header reduction techniques have various selected TVA-enabled object types in common. This is quantified in Table 4.1 where the number of TVA-enabled types are shown for offline and online type selection as well as the number of object types in common between offline and online type selection.

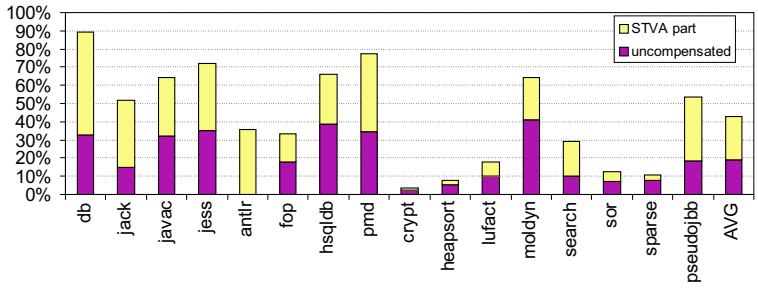

Figure 4.8: Memory usage overhead of 64-bit mode compared to 32-bit mode and the part thereof that is reduced through STVA under the online no-header model.

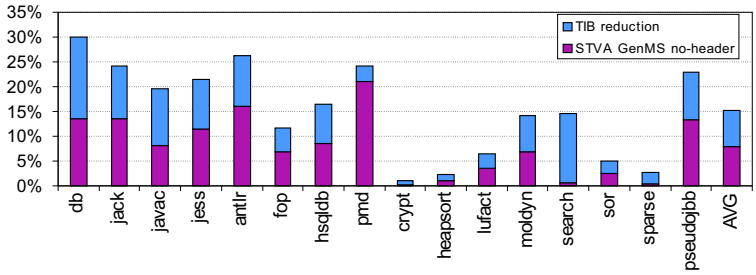

Figure 4.9: The reduction in allocated bytes partitioned by TIB pointer compression and online no-header STVA for the GenMS collector.

To validate the effect of STVA on the reduction of the memory usage overhead introduced by 64-bit mode, we present in Figure 4.8 the increase in the number of bytes allocated when going from 32-bit mode to 64-bit mode and we mark the part that STVA under the online no-header model reduces thereof. On average, more than half of the memory usage overhead is reduced and even a reduction of up to 100% is seen for antlr.

Reduction through TIB pointer compression versus STVA

Figure 4.9 shows the reduction in allocated bytes partitioned by (i) the TIB pointer compression technique and (ii) the no-header STVA object model. We observe that approximately half the memory savings comes

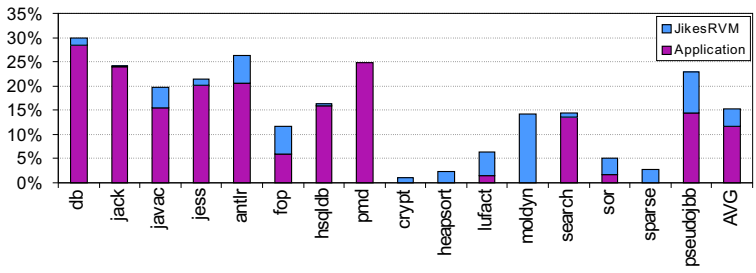

Figure 4.10: Accounting the overall memory reduction to application and VM objects; this graph assumes the GenMS garbage collector and the online no-header STVA object model.

from TIB pointer compression that is applied to all objects; the other half comes from the no-header STVA object model that can be applied only to TVA-enabled objects.

Reduction in application objects versus VM objects

As mentioned in previous chapters, Jikes RVM is a VM that is written in Java. As a consequence, STVA also applies to objects allocated by the VM and thus, the results presented account for applying STVA to both VM objects and application objects. Other VMs that are not written in Java on the other hand, may not get similar benefits from STVA as what is presented here. In order to quantify the impact of our experimental setup using Jikes RVM, we classify the objects as VM and application objects and then compute the amount of memory reduction for VM and application objects separately. Classifying objects as VM and application objects is done by the technique described in section 2.4.1.

Figure 4.10 quantifies the amount of memory reduction for the application objects and VM objects. The important observation here is that the most significant part of the overall 15.2% memory reduction is obtained through the application objects (11.6% on average); about 3.6% is accounted for by VM objects. These results show that other VMs that are not written in Java can also significantly benefit from implementing STVA for reducing overall memory usage.

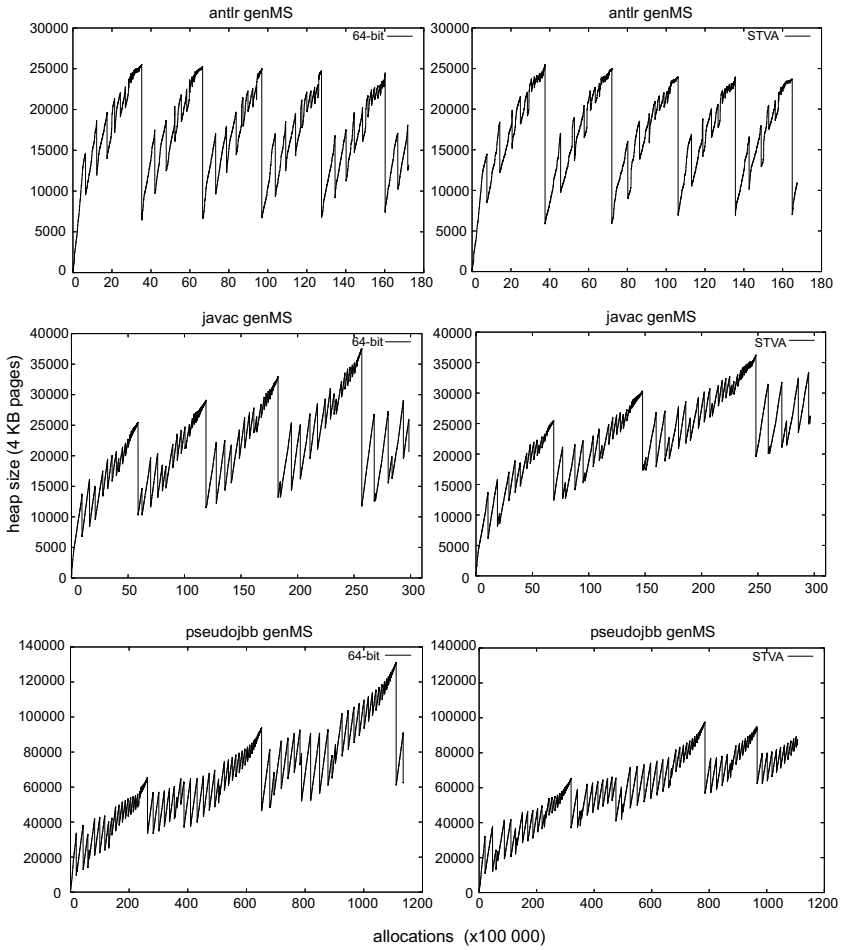

Figure 4.11: Heap growth for GenMS collector as a function of time (measured per allocation site) for antlr (top), javac (middle) and pseudojbb (bottom). The graphs on the left show the results for a traditional VM; the graphs on the right show the results for a STVA-aware VM.

Reduction in in-use memory pages

Figure 4.11 shows the heap size counted as the number of pages in use on the vertical axis as a function the number of allocations on the horizontal axis for antlr, javac and pseudojbb, respectively. The curves in these graphs increase as memory gets allocated until a garbage collection is triggered after which the number of used pages drops to the amount of reachable data at that point. This explains the shape of these

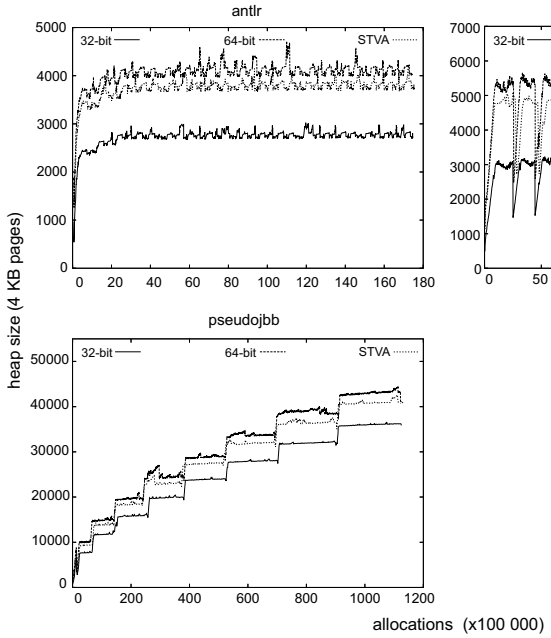

Figure 4.12: Maximum reachable bytes (measured in 4KB pages) as a function of time (measured per allocation site) for three benchmarks, antlr (top left), javac (top right), pseudojbb (bottom) for 32-bit and 64-bit processing and for STVA under the online no-header model.

graphs. There are two important observations to be made from these graphs. First, since STVA reduces the amount of allocated bytes per allocation, garbage collections get delayed — the STVA curve is shifted to the right compared to the original Jikes RVM curve. In other words, fewer garbage collections are required. Second, when garbage is collected, the number of pages in use for STVA can drop below the number of pages in use for the original Jikes RVM. The reason is that the amount of reachable bytes is smaller under STVA because of the space-efficient STVA object model. If we compare the graphs for STVA with the graphs in chapter 2 (Figure 2.2), we see that the shape of the graph shifts back towards the shape it has on the 32-bit platform. This of course is the behavior we expect, since the original shift towards the 64-bit graphs was caused by the increased memory usage and the goal of STVA is to reduce memory usage again.

Reduction in maximum reachable bytes

Figure 4.12 shows the maximum reachable bytes (in pages) as a function of time (measured in the number of allocations) on the horizontal axis for antlr, javac and pseudojbb, respectively. We start with the 32-bit and 64-bit graphs from Figure 2.4, and add now curves for the maximum reachable bytes for STVA under the online no-header model. We observe a clear reduction in the maximum reachable bytes for STVA compared to the 64-bit base case for all three benchmarks: between 5% and 7% for pseudojbb, more than 7% for antlr and more than 10% for javac. These reductions are less than the reduction in total allocated bytes (Figure 4.7), so we can conclude for these three benchmarks, that although the STVA technique also significantly reduces the memory overhead of long-lived objects, it primarily works on the objects with shorter lifetimes.

Impact on garbage collection

Tables 4.2 and 4.3 give an overview of the actual number of garbage collections performed. If we compare the number of GCs of STVA to the number of GCs in the base case, we observe an overall decrease of the number of collections. On average 18.9% and 16.9% fewer collections are performed for minor and major collections, respectively. This reduction comes from the reduced number of pages in use as discussed above. Figure 4.13 shows the speedup of the garbage collector for the online and offline configurations. For one configuration, the GenCopy no-header configuration, we observe a general slowdown in GC time (15.9%). Most benchmarks show a (small) speedup in total garbage collection time for the small-header configuration. The highest speedup is 132% for jack. The reason why the general significant speedup of the total garbage collection time is small, while there is a significant reduction in number of GCs, is that extra work needs to be done during each garbage collection to determine which object model applies to a scanned object and to access the side arrays for the no-header configurations.

4.8.3 Performance

Figures 4.14 and 4.15 show the speedup in terms of overall performance for the offline and online reduced header object models, respectively.

Table 4.2: Number of minor and major GCs under the GenMS collection scheme for the base 64-bit VM and for the small header and the no-header STVA-aware VMs.

	minor GCs			major GCs		
Benchmark	base	small hdr	no hdr	base	small hdr	no hdr
db	62.7	31.8	31.4	2.0	2.0	2.0
jack	184.0	91.1	93.4	2.0	0.0	0.1
javac	120.7	115.9	117.1	3.5	3.0	3.0
jess	75.9	55.3	50.7	0.0	0.0	0.0
antlr	106.7	90.6	89.1	5.0	5.0	4.0
fop	25.3	13.1	13.6	0.5	0.0	0.0
hsqldb	58.3	74.3	65.0	5.0	4.0	4.0
pmd	275.6	257.4	218.6	13.0	10.9	9.5
crypt	1.0	1.0	1.0	0.0	0.0	0.0
heapsort	1.0	1.0	1.0	0.0	0.0	0.0
lufact	1.0	1.0	1.0	0.0	0.0	0.0
moldyn	1.0	1.0	1.0	0.0	0.0	0.0
search	18.9	16.0	16.0	0.0	0.0	0.0
sor	1.0	1.0	1.0	0.0	0.0	0.0
sparse	1.0	1.0	1.0	0.0	0.0	0.0
pseudojbb	116.9	100.5	100.0	4.1	4.0	4.0

Table 4.3: Number of minor and major GCs under the GenCopy collection scheme for the base 64-bit VM and for the small header and the no-header STVA-aware VMs.

	minor GCs			major GCs		
Benchmark	base	small hdr	no hdr	base	small hdr	no hdr
db	32.8	37.0	33.4	3.0	3.0	3.0
jack	193.9	144.6	129.4	2.1	1.2	1.1
javac	115.1	104.6	104.6	5.9	5.4	5.1
jess	167.5	119.5	153.9	1.0	0.2	0.1
antlr	104.7	66.5	70.5	5.9	5.0	5.0
fop	15.2	13.5	14.8	1.0	1.0	1.0
hsqldb	60.1	54.4	52.9	6.0	5.0	5.0
pmd	214.7	212.1	198.6	13.9	12.7	12.7
crypt	1.0	1.0	1.0	0.0	0.0	0.0
heapsort	1.0	1.0	1.0	0.0	0.0	0.0
lufact	1.0	1.0	1.0	0.0	0.0	0.0
moldyn	1.0	1.0	1.0	0.0	0.0	0.0
search	21.0	14.9	17.4	0.0	0.0	0.0
sor	1.0	1.0	1.0	0.0	0.0	0.0
sparse	1.0	1.0	1.0	0.0	0.0	0.0
pseudojbb	112.2	91.5	95.6	6.2	5.9	6.0

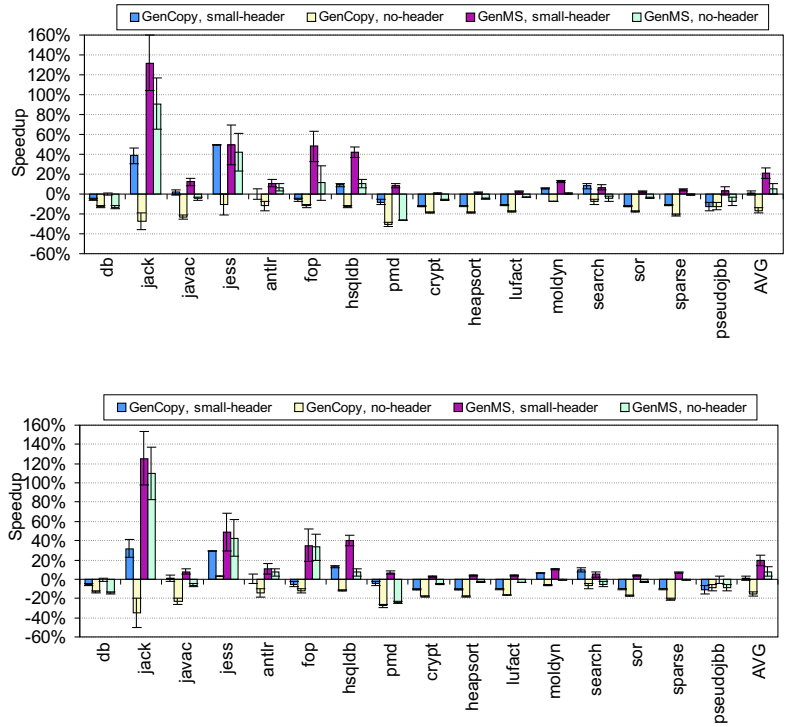

Figure 4.13: Speedup of the garbage collector for offline and online header reduction.

Data is shown for the small-header and no-header object models as well as for the GenCopy and the GenMS collectors. These figures show speedups along with the 95% confidence intervals. The offline reduced header object models are obtained from a cross-validation setup, i.e., we use profile inputs for selecting the TVA-enabled types, and we use reference inputs for reporting speedups. We set MRT and LLMRT to 0.1% in these experiments based on the results of the feasibility study obtained in section 4.8.1.

We observe that for some benchmarks, STVA results in a statistically significant performance degradation. This suggests that the run time overhead introduced by STVA has a larger impact on overall performance than the reduction in memory footprint for these benchmarks. The performance degradation that we see is generally smaller than 5%, For a number of benchmarks, we observe larger performance degradations, but for only a few collector and object model configurations,

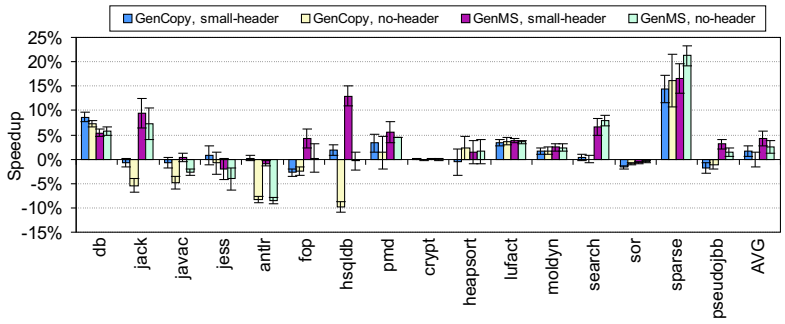

Figure 4.14: Speedups along with the 95% confidence intervals for offline header reduction. The MRT and LLMRT thresholds are set to 0.1%.

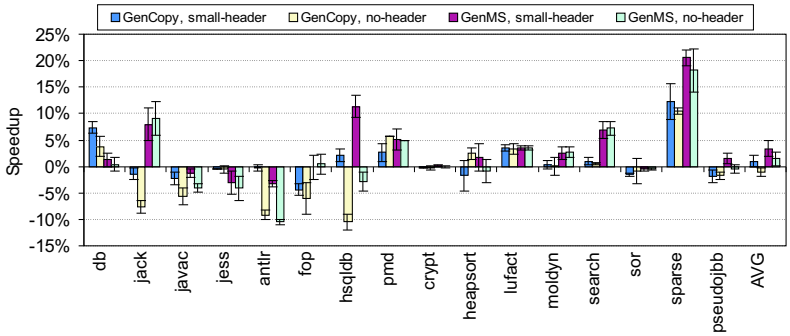

Figure 4.15: Speedups along with the 95% confidence intervals for online header reduction.

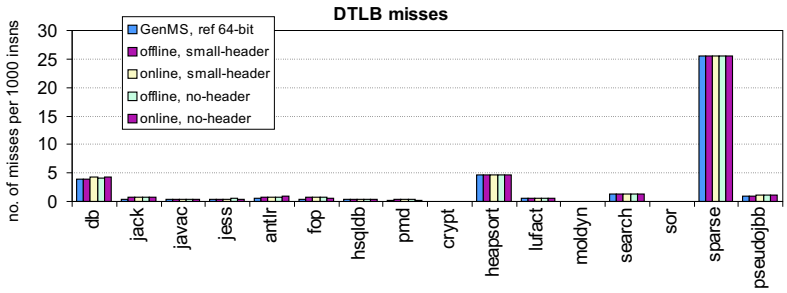

Figure 4.16: The number of D-TLB misses per 1000 instructions in the reference run for the GenMS garbage collector.

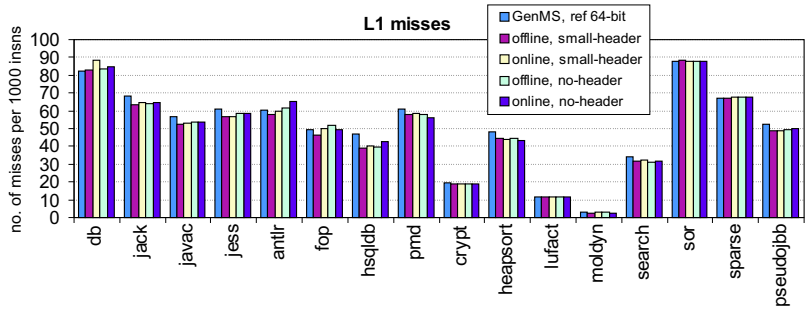

Figure 4.17: The number of L1 D-cache misses per 1000 instructions in the reference run for the GenMS garbage collector.

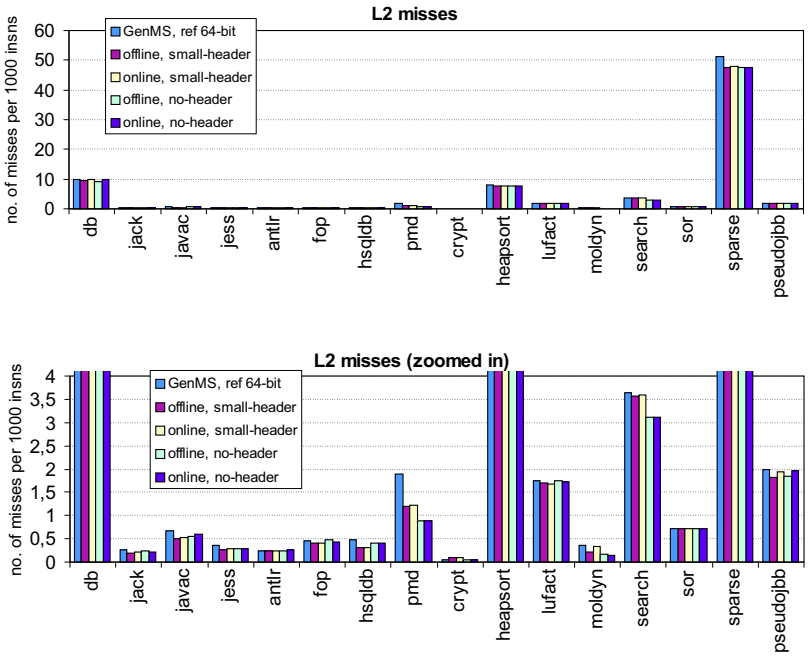

Figure 4.18: The number L2 D-cache misses per 1000 instructions in the reference run for the GenMS garbage collector. The only difference between the first and second graph is the scale of the vertical-axis.

while other configurations even show performance improvements for the same benchmarks, e.g., jack, javac, antlr and hsqldb. This indicates that these benchmarks are very sensitive to the placement of objects on

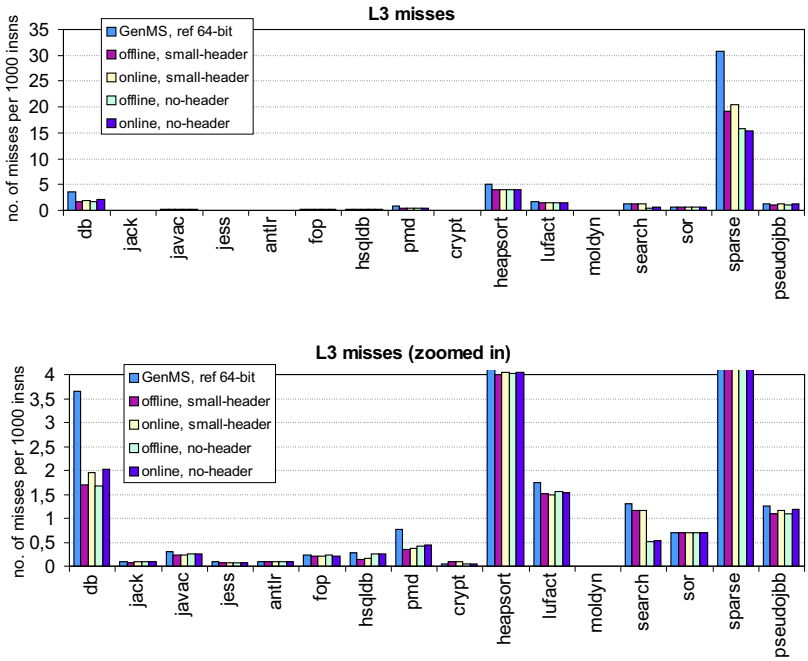

Figure 4.19: The number L3 D-cache misses per 1000 instructions in the reference run for the GenMS garbage collector. The only difference between the first and second graph is the scale of the vertical-axis.

the heap. A number of benchmarks show a significant performance improvement: db (7%), pmd (5%), lufact (4%), moldyn (3%), sparse (up to 20%) and jack and search for the GenMS collector (up to 9%), and also hsqldb for the GenMS small-header configuration (up to 13%). For all remaining benchmarks, STVA has no statistically significant impact on overall performance. In conclusion, the space-efficient object models do not have a negative impact on performance for most of the benchmarks and a couple of benchmarks even show a significant speedup.

4.8.4 Cache and TLB performance

We now study the impact of STVA on cache and TLB performance in more detail using hardware performance counters. Figure 4.16 quantifies the number of D-TLB misses, Figures 4.17, 4.18 and 4.19 quantify the number of D-cache misses for the L1, L2 and L3 level D-cache,

respectively. The number of misses are measured per 1000 instructions in the reference run for the GenMS garbage collector; we obtained similar results for the GenCopy garbage collector. The graphs for the L2 and L3 cache levels are shown twice to magnify the results for benchmarks with a small number of misses per 1000 instructions. We observe that for a few benchmarks the number of D-TLB misses slightly increases due to the increased memory fragmentation because of STVA. The number of cache misses typically decreases, especially for the larger L2 and L3 caches; the reason is the reduced memory usage. On average the number of cache misses are reduced by 2.7%, 11.1% and 16.5% for L1, L2 and L3 misses, respectively. For some benchmarks such as db and sparse, the number of L3 misses decreases by 50%. This large decrease in L3 misses explains the speedup results reported in Figures 4.14 and 4.15.

We acknowledge that our TVA technique might have a bad impact on spatial locality of objects of different types that are frequently accessed together. However, our cache performance and overall performance results show that the locality improvements introduced by TVA, due to less memory usage and better cache usage, overcompensate for this loss of spatial locality. Further exploration of techniques for reducing this spatial locality loss, might thus further improve our TVA technique. In particular, architectures that support virtual address aliasing, could implement our TVA technique without interfering with the order in which objects are layed out on the heap. Objects of different types allocated on the same physical page, can than be assigned a (different) virtual address of a different typed virtual address segment. This way, spatial locality is not lost and fragmentation no longer exists (on the physical page level).

4.8.5 STVA versus TVA

As mentioned before, one contribution of this work is to show that applying TVA to a selected number of object types (i.e., STVA) results in better performance than applying TVA to all object types. This is clearly shown in Figure 4.20 where STVA is compared to TVA. TVA performs fairly well in general and achieves similar performance as STVA for many benchmarks. However, for a number of benchmarks, TVA results in significant performance degradations compared to STVA. This is the case for javac, hsqldb, pmd, lufact, sparse and pseudojbb. The reason for this is memory fragmentation: objects from not frequently allocated

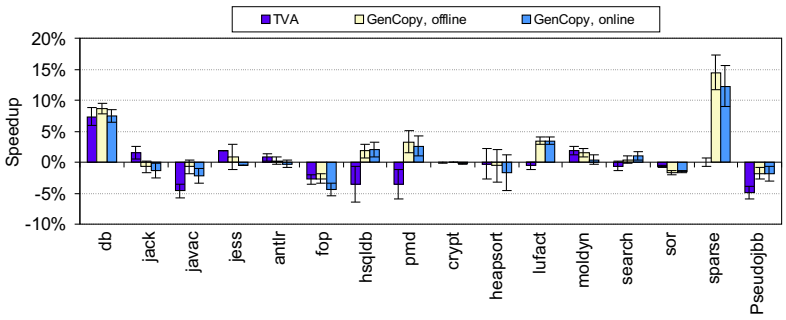

Figure 4.20: Comparing STVA to TVA in terms of speedup for the GenCopy garbage collector.

object types will all pollute memory chunks that all get sparsely filled. All in all, we conclude that implicit typing on selected object types outperforms implicit typing on all object types.

4.9 Related work

Adl-Tabatabai et al. [3] address the increased memory requirements of 64-bit Java implementations by compressing 64-bit pointers to 32-bit offsets. They apply their pointer compression technique to all pointers, including the TIB pointer and the forwarding pointer in the object header. By compressing the TIB pointer and the forwarding pointer in the object header, they can actually reduce the size of the object header from 16 bytes (for non-array objects) to only 8 bytes. There are three key differences with our approach. First, we eliminate the TIB pointer completely from the object header for TVA-enabled objects; they only compress the TIB pointer. The second difference between Adl-Tabatabai et al.'s approach and our proposal is that we do not need to compress and decompress the TIB pointer. We compute the TIB pointer from the object's virtual address. And finally, the approach by Adl-Tabatabai et al. limits applications to a 32-bit address space. As such, applications that require more than 4 GB of memory cannot benefit from pointer compression. STVA and TIB pointer compression do not suffer from this limitation. The only assumption we make in our proposal is that we do not need more than a 32-bit virtual address space for holding

type information, however, it is highly unlikely that this would ever be needed in practice.

Bacon et al. [8] present a number of header compression techniques for the Java object model on 32-bit machines. They propose three approaches for reducing the space requirements of the TIB pointer in the header: bit stealing, indirection and the implicit type method. Bit stealing uses the least significant bits from a memory address (which are typically zero) for other purposes. The main disadvantage of bit stealing is that it frees only a few bits. Indirection represents the TIB pointer as an index into a table of TIB pointers. The disadvantages of indirection are that an extra load is needed to access the TIB pointer, and that there is a fixed limit on the number of TIBs and thus the number of object types that can be supported. Bit stealing and indirection still require a condensed form of a TIB pointer to be stored in the header. The approach that we propose however, has the advantage over these two approaches to completely eliminate the TIB pointer from the object header.

The third header compression method discussed by Bacon et al. is called the implicit type method. The general idea behind implicit types is that the type information is part of the object's virtual address. In fact, there are number of ways of how to implement implicit typing. A first possibility is to have a type tag included in the pointer to the object. The type tag is then typically stored in the most-significant or least-significant bits of the object's virtual address. By consequence, obtaining the effective memory address requires masking the object's virtual address. Storing the type tag in the most-significant bits of the object's virtual address usually restricts the available address space. Storing the type tag in the least-significant bits of the object's virtual address on the other hand, usually forces objects to be aligned on multiple byte boundaries. A second approach is to use the type tag bits as a part of the address. By doing so, the address space gets divided into several distinct regions where objects of the same type get allocated into the same region. This is similar to the TVA implementation that we use in this chapter.

Another approach is the Big Bag of Pages (BiBOP) approach proposed by Steele [65] and Hanson [36]. In BiBOP, the type tag serves as an index into a table where the type is stored. BiBOP views memory as a group of equal-sized segments. Each segment has an associated type. An important disadvantage of BiBOP typing is that the type tag that

is encoded in the memory address serves as an index in a table that points to the object's TIB. In other words, an additional indirection is needed for accessing the TIB. Dybvig et al. [27] propose a hybrid system where some objects have a type tag in the least-significant bits and where other objects follow the BiBOP typing.

The typed virtual memory addressing that we propose here in this work differs from this prior work on typed virtual addressing in the following major ways. First, we propose to apply implicit typing to selected object types only; previous work applied implicit typing to all object types. Applying implicit typing to all object types results in significant memory fragmentation. We argue and show how to make a good selection on what objects to apply the implicit type method. Second, although previous work already describes the implicit type method, we could not find any papers actually evaluating it or comparing it to memory systems without typed virtual addressing. In this dissertation, we propose a practical method of how to implement the implicit typing method for 64-bit Java VM implementations and, in addition, we quantify the performance and memory usage impact of STVA and compare that to traditional VM implementations without STVA.

Shuf et al. [63] propose the notion of prolific types versus non-prolific types. A prolific type is defined as a type that has a sufficiently large number of instances allocated during a program execution. In practice, a type is called prolific if the fraction of objects allocated by the program of this type exceeds a given threshold. All remaining types are referred to as non-prolific. Shuf et al. found that only a limited number of types account for most of the objects allocated by the program. They then propose to exploit this notion by using short type pointers for prolific types. The idea is to use a few type bits in the status field to encode the types of the prolific objects. This way, the TIB pointer field can be eliminated from the object header. The prolific type can then be accessed through a type table. A special value of the type bits, for example all zeros, is then used for non-prolific object types. Non-prolific types still have a TIB pointer field in their object headers. A disadvantage of this approach is that the number of prolific types is limited by the number of available bits in the status field. In addition, computing the TIB pointer for prolific types requires an additional indirection. Our STVA implementation does not have these disadvantages. The advantage of the prolific approach is that the amount of memory fragmentation is limited since all objects are allocated in a single segment, much as in traditional VMs. The STVA implementation that we

propose could be viewed of as a hybrid form of the prolific approach and the implicit typed methods discussed above; we apply implicit typing to prolific types.

Other proposed techniques, not specifically targeted towards 64-bit systems, like object inlining [51], also save space although the objective is to reduce pointer chasing. Object inlining saves the header space of the inlined object and the pointer field in the referencing object. Object inlining requires a uniqueness constraint on pointer fields in order to be applicable.

4.10 Conclusion

This chapter proposed eliminating the header from the 64-bit Java object model through Selective Typed Virtual Addressing (STVA). The idea of STVA is to apply typed virtual addressing (TVA) or implicit typing to a selected number of object types. TVA means that the object type is encoded in the object's virtual address. We apply TVA selectively, hence the name Selective TVA, on object types that are frequently allocated. The end result is that the header can be eliminated completely from the object header. The TIB pointers are stored in the TIB space and the status field information is stored in side arrays. Accessing the appropriate TIB pointer and status field is done through offsets computed from the object's virtual address. For the objects on which we do not apply TVA, we compress the TIB pointer from 64-bit to 32-bit.

We evaluated our newly proposed space-efficient Java object model in a 64-bit Java VM implementation, namely Jikes RVM, on an AIX IBM POWER4 machine. Our results show that the space-efficient object model yields a reduction in the number of allocated bytes by 15% on average (and up to 35% for some benchmarks). Half the reduction comes from STVA; the other half comes from TIB pointer compression. The reduction of allocated bytes also impacts the D-cache misses: 2.7%, 11.1% and 16.5% fewer misses for the L1, L2 and L3 cache level, respectively. In terms of performance, the space-efficient Java object model generally does not affect performance in a statistically significant way, however, some benchmarks exhibit significant overall speedups (up to 20%).

Chapter 5

Conclusion

Science may set limits to knowledge,
but should not set limits to imagination.
Bertrand Russell

In this chapter we summarize the conclusions that can be drawn from this dissertation and we highlight some research topics that could be investigated as future work.

5.1 Summary

This thesis investigated and improved the behavior of Java applications on 64-bit general-purpose computer system. **64-bit technology cannot be stopped**: it has been used in high-end servers for over fifteen years now and currently it is incorporated in all general-purpose personal computer systems. History has taught us that customers are a bit doubtful towards sudden disruptive changes in computer hardware designs that do not support upward compatibility. That is why hardware manufacturers often build systems that are backward compatible, and why most of the 64-bit computer systems have some form of 32-bit compatibility mode. Due to the co-existence of two modes, OS-developers, compiler writers and people building other software tools, are given the time to adapt to the newest mode. But eventually the general-purpose computer systems will drop the compatibility mode.

As with most things, also 64-bit technology, when compared to 32-bit technology, has certain advantages as well as disadvantages. **The**

largest disadvantage is the increased memory usage and the biggest advantage is the availability of extra 64-bit instructions. The benefit perceived by the user depends on the application. If an application cannot benefit from the extra 64-bit instructions, it will almost certainly suffer from the negative impact of the increased memory usage. The larger the negative impact, the more reluctant consumers will be to stop using the 32-bit compatibility mode. Although the effects of increased memory usage can be reduced at the cost of increasing physical memory, this is not a fundamental solution. Hence, in this dissertation we studied and improved the increased memory usage due to 64-bit computing. We performed this study in a Java environment, because applications written in OO-languages (as is the case for Java) are very sensitive for the increased memory usage problem due to the transition from 32-bit to 64-bit computing.

We observed that **objects are on average 45.3% larger** in a 64-bit VM than in a 32-bit VM. We identified four reasons for this increase: (i) larger pointers in object data fields, (ii) the object's header doubles in size, (iii) additional padding for intra-object alignment and (iv) extra space between objects due to inter-object alignment. The increased memory usage puts more pressure on the GC system. On average, 64.5% more time is spent during GC while 60.1% more minor collections and 64.8% more major collections are performed in the 64-bit VM than in the 32-bit VM, for the setup used in this dissertation.

In order to reduce the pressure on the memory system we introduced two memory reduction techniques. The first technique, **Object-Relative Addressing**, compresses pointers inside object data fields as a 32-bit offset from the object's base address. This technique yields a reduction in the number of allocated bytes of more than 10% for many benchmarks. In addition, ORA also reduces the number of minor and major GCs by 10.6% and 17.8% on average, respectively. In order to apply a pointer compression scheme correctly, a number of issues need to be dealt with. The most important issues are (i) to decompress the pointer value correctly at all times, (ii) to choose an appropriate and efficient representation for special pointer values and (iii) to update compressed pointer values correctly when objects are moved.

For the first issue, being able to decompress the pointer value correctly at all times, we created a Long Address Table (LAT) that holds incompressible pointers in case a pointer cannot be compressed by the ORA scheme. This table is dynamically extensible if needed and ob-

solete entries are removed during garbage collection. We use the least significant bit of the compressed data to disambiguate between table entries and truly compressed pointers. We proposed two mechanisms to determine whether a value at hand is a truly compressed pointer or not. The first mechanism is a simple bit test. If the bit is set, we access the LAT, otherwise we decompress the fast way. As a second mechanism, we initially assume that the value is a truly compressed pointer. Only at the event that a pointer cannot be compressed, we patch all code locations that potentially load that value to take the bit test mechanism.

For the second issue, to choose an appropriate and efficient representation for special pointer values, we use 32 zero bits as the `this` pointer, because our ORA decompression scheme automatically decompresses 32 zero bits to the `this` pointer. In order to represent the `null` value, we choose the value that gets decompressed by our ORA decompression scheme to a value with the 32 least significant bits set to zero. As a consequence, there is no longer a single `null` value.

To handle the third issue, update compressed pointer values correctly when objects are moved, we keep track of both the original pointer as well as the pointer to the copied object. As such, we do not have to update the compressed pointer twice (on movement of either the referencing or referenced object), but only once after both the referencing and referenced objects are (possibly) moved.

The second approach which we propose to reduce the memory usage of applications, is enabled by the **Selective Typed Virtual Addressing** technique. TVA encodes the object type in the object's virtual address by allocating objects of a given type in a designated memory segment. We showed that applying TVA to all object introduces memory fragmentation and hence performance loss, and hence we make a selection as to which object types TVA is applied. Hence, the name Selective Virtual Typed Addressing (STVA). STVA reduces memory demands with 15% on average and reduces the number of minor and major garbage collections by 18.9% and 16.9%, respectively.

First, we proposed an offline selection technique for STVA. Through a profiling run of the application we collect the number of allocated objects and the number of collected objects for each object type and the allocated objects' sizes. From this information the amount of space can be computed that can be gained for each type when applying TVA. We selected those object types for STVA that give the largest memory

savings during allocation/collection. Subsequently, we proposed an online selection technique, based on method sampling. The VM uses a sampling technique to identify frequently executed methods, so called hot methods. These methods are scheduled for re-compilation and re-optimization. We select all object types allocated in these hot methods to be candidates for STVA, because these types are likely to be allocated frequently.

When applying STVA, we considered two variants: one that reduces the object header from 16 to 4 bytes and one that completely removes the object header. The 4 bytes left in the former (the small-header object model) contain space for information about hashing, locking and GC. In the no-header object model we move this information out of the object header. We apply an alternative locking scheme and the hash state and GC bits are move into side-arrays. Side-arrays are meta-data that we allocate on every two pages of memory: one byte for every object on those pages. Because those pages contain equally-sized objects, side-array access gets simplified.

5.2 Future work, a perspective

5.2.1 Embedded systems

The research in this dissertation was conducted on a general-purpose computer system. In the embedded computer world, resources are scarce. For example most embedded systems have strict memory constraints. There are only a few embedded domains that embraced 64-bit technology, like digital signal processing and 3-D gaming. This is because these domains are targeted towards applications that can benefit from the extra 64-bit instructions. Manufacturers of other embedded systems, which cannot exploit these benefits, will not use 64-bit technology, because then they put extra pressure on their memory system.

A future research area would be to investigate how the techniques presented in this dissertation could be implemented in the (32-bit) embedded world. Different design choices will need to be made. We will start with a brief discussion concerning the ORA technique. Using 16 bits as a relative offset, limits the reachability to 64 KB through a relative pointer. This is very limited, and thus trying to apply ORA (and avoiding large overhead) in such circumstances would (probably) require that the ORA technique gets applied to certain small groups of

strongly connected objects. This requires a more specific analysis of the connectivity between objects in order to allow the memory allocator and the garbage collector to lay out these connected groups together in memory. An advantage in the embedded world is that extra hardware instructions could be added to fasten compression/decompression and some hardware support could be added to manage the representation of pointers that are further away than 64 KB from each other.

Concerning the STVA technique, in a 32-bit embedded system, the fragmentation impact of the STVA technique will be more significant because of the stricter memory constraints. Also, the encoding of the type in a 32-bit pointer needs to be efficient, in order to keep a type's memory chunk large enough. As with the ORA technique, hardware support could also assist STVA, namely in efficiently encoding/decoding the type and in distinguishing STVA-enabled objects from STVA-disabled objects.

5.2.2 Creating ORA regions

As explained in chapter 3, we envision ORA to be used in conjunction with a memory allocator and a garbage collector that strive at limiting the number of inter-object references that cross the 32-bit address range. A possible way to build such a memory manager, is to use techniques similar to object collocation [34], connectivity-based memory allocation and collection [38, 39], region-based systems [22, 57], etc. First, an analysis step is needed to determine points-to information. Next, a heuristic needs to be designed to segregate objects into memory regions. A good heuristic would divide objects across regions in such a way that no (or almost no) inter-region pointers exist, and hence ORA could be applied to all intra-region pointers. For inter-region pointers one may want to ORA-disable them in order not to impose any overhead. Regions can be garbage collected in order to keep them compact. In this case, the garbage collector needs to be aware of the region an object belongs to. Future research will need to examine how objects can be segregated efficiently in a way that suites ORA. Current points-to analyses are often performed statically. In order to get more accurate points-to information, an interesting research area is to investigate how these techniques can be done dynamically with as less overhead as possible.

5.2.3 Combining ORA and STVA

ORA and STVA are both memory compression techniques at allocation time and each of them apply to a separate part of the object: ORA applies to the pointers in the data part and STVA applies to the header part. This makes them excellent candidates to be implemented together as complementary techniques. However, this requires additional research. Both techniques take a different approach to manage object placement. The STVA technique on the one hand, segregates the memory in typed segments, thus objects of the same type are allocated close to each other, and hence objects of different (STVA) types are allocated far from each other. The ORA technique on the other hand, wants objects that are connected to be allocated in close proximity (which may have different types).

These strategies seem to be in conflict at first glance, but they do not need to be *per se*. The STVA memory segments do not need to be contiguous as is the case in our implementation. It would be possible to define memory regions according to the ORA technique, and to define typed memory segments within each region. One STVA memory segment would then reappear in each ORA memory region: they are interleaved. Another possibility is to apply both techniques to only a subset of the objects in such a way that they no longer conflict.

Yet another possibility might be to keep both implementations as they are: STVA allocates objects of a certain type in separate segments in virtual memory, and all non-STVA objects get allocated in ORA regions. Now a priori, ORA regions and STVA segments will not be in close proximity, so short pointers from one to the other would always need to go through the LAT. To solve this, one of the least significant bits of a pointer can be used through bit-stealing to indicate that this short pointer escapes to the STVA segment or vica versa. In case such a short pointer escapes from an ORA region to a STVA segment, the actual type segment can be derived from the (static) type of that pointer. The other way around, from a STVA segment to an ORA region, one could favor one ORA region, all other ORA regions would still need to use the normal slow path through the LAT.

Future research on this topic will need to investigate which possibility (or a combination) is best pursued. All by all, (small) objects without references are always good candidates for STVA, e.g., objects of type Integer.

In terms of combined memory reductions, it is expected that the combination of STVA and ORA will lead to a reduction in allocated bytes of about the sum of the individual byte reductions, possibly more. This is because they focus their effort on a different part of the object (the header part versus the body part), and because the combination can have extra inter-object alignment gains over both techniques. For the no-header STVA object model, a reduction of the inter-object alignment overhead is only possible for objects without references in their data fields. The ORA technique can only reduce the inter-object alignment overhead, if it first compressed at least one pointer. The combination of both techniques can therefore not perform worse in terms of byte reduction, only equally well or better as both techniques separately, added together. An example of the latter is, for instance, when considering an object with two integer fields and one reference field. STVA can remove the header, and ORA can compress one pointer, but will instead introduce an extra 4 bytes inter-object alignment. The combination of both would no longer need to introduce the extra four bytes because all the field sizes after applying both techniques are 4 bytes in size and hence do not require 8-byte alignment. To present an indication about the expected reduction of the memory overhead of 64-bit compared to 32-bit, after the combination of the ORA and STVA techniques, Figure 5.1 shows once again the extra memory overhead of 64-bit mode compared to 32-bit mode. We mark the parts thereof that get reduced by the ORA and STVA techniques separately. On average about one fifth of the 45% memory overhead is still not reduced. This can be explained by the fact that (i) our techniques are not applied to all the objects, (ii) the extra memory used by data structures needed by our techniques and (iii) the code increase due to implementing our techniques (the code compiled by the VM also end up in the data heap). For some benchmarks, we even see an overcompensation, e.g., 14% for antlr. This can be the case if most objects are covered by our techniques, and because the STVA technique can remove more bytes from the header than were added by the transition from 32-bit to 64-bit mode.

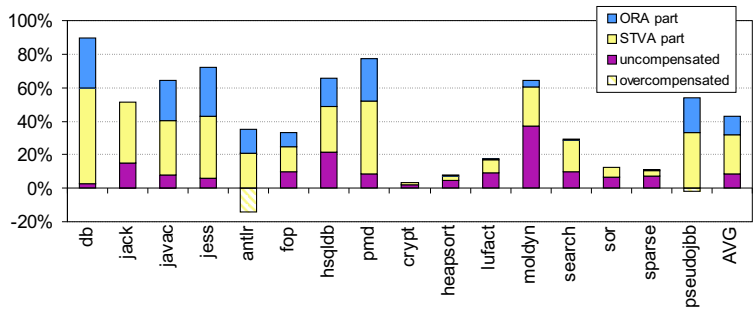

Figure 5.1: Memory usage overhead of 64-bit mode compared to 32-bit mode and the parts thereof that are reduced through ORA and STVA.

Bibliography

[1] The AMD x86-64 architecture programmers overview. Technical Report 24108C, Advanced Micro Devices, 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_overview.pdf.

[2] Intel Itanium processor family reference guide. Technical Report 254318-003, Intel Corporation, 2004.

[3] A.-R. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. T. Lewis, B. R. Murphy, and J. M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 100–110. IEEE Computer Society, March 2004.

[4] O. Agesen. Space and time-efficient hashing of garbage-collected objects. *Theory and Practice of Object Systems*, 5(2):119–124, 1999.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[6] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 59–68, June 2003.

[7] A. W. Appel. A runtime system. Technical Report CS-TR-220-89, Princeton University, Computer Science Department, May 1989.

[8] D. F. Bacon, S. J. Fink, and D. P. Grove. Space- and time-efficient implementation of the Java object model. *Sixteenth European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science*, 2374:111–132, 2002.

[9] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, June 1998.

[10] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proc. of 36th International Conference on Microarchitecture (MICRO36)*, pages 191–201. IEEE Computer Society, December 2003.

[11] S. Behling, R. Bell, P. Farell, H. Holthoff, F. O'Connel, and W. Weir. *The POWER4 Processor Introduction and Tuning Guide*. Redbooks. IBM Corporation, International Technical Support Organization, 2001.

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the 2004 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 25–36. ACM Press, June 2004.

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*. ACM Press, October 2006.

[14] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. Garbage collection hints. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC), LNCS 3793*, pages 233–348. Springer Verlag, November 2005.

[15] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. GCH: Hints for triggering garbage collections. *Transactions on High-Performance Embedded Architectures and Compilers*, 1(1):52–72, June 2006.

[16] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the seventh IEEE International Symposium on High-Performance Computer Architecture*, pages 229–240. IEEE Computer Society, January 2001.

[17] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[18] M. Chapman, I. Wienand, and G. Heiser. Itanium page tables and TLB. Technical Report Technical Report UNSW-CSE-TR-0307, University of New South Wales, Sydney, Australia, May 2003.

[19] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting frequent field values in Java objects for reducing heap memory requirements. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 68–78. ACM Press, 2005.

[20] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Field level analysis for heap space optimization in embedded Java environments. In *International Symposium on Memory Management (ISMM)*, pages 131–142. ACM Press, 2004.

[21] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 282–301. ACM Press, October 2003.

[22] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM '04: Proceedings of the 4th International Symposium on Memory Management*, pages 85–96. ACM Press, 2004.

[23] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM Press, May 1999.

[24] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the first International Symposium on Memory Management (ISMM)*, pages 37–48. ACM Press, October 1998.

[25] Advanced Micro Devices. x86-64 technology white paper. August 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_wp.pdf.

[26] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the 13th European Conference for Object-Oriented Programming (ECOOP)*, pages 92–115. Springer, June 1999.

[27] R. K. 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, 1994.

[28] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–186. ACM Press, October 2003.

[29] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85. IEEE Computer Society, June 2005.

[30] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 270–287. ACM Press, October 2004.

[31] R. Y. Gevai. Porting code to Intel EM64T-based platforms. Technical Report 254740, Intel Software and Solutions Group.

[32] P. N. Glaskowsky. Athlon 64 moving to mass market. *Microprocessor report*, January 2004.

[33] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, Boston, MA, USA, 1997.

[34] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 237–250. ACM Press, October 2004.

[35] T. R. Halfhill. AMD and Intel harmonize on 64. *Microprocessor report*, March 2004.

[36] D. R. Hanson. A portable storage management system for the icon programming language. *Software-Practice and Experience*, 10(6):489–500, 1980.

[37] J. L. Hennessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, New York, USA, 3rd edition, 2003.

[38] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 359–373. ACM Press, 2003.

[39] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management (ISMM)*, pages 36–39. ACM Press, jun 2002.

[40] R. Jones and R. Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, Inc., 1996.

[41] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, page 8. Springer Verlag, July 2006.

[42] T. Kaehler and G. Krasner. Loom: large object-oriented memory for smalltalk-80 systems. pages 251–270, 1983.

[43] M. Karlsson, K. E. Moore, E. Hagersten, and D. A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the Ninth IEEE International Symposium on High-Performance Computer Architecture*, pages 217–228. IEEE Computer Society, February 2003.

[44] K. Kersey. Intel's new platform versus AMD's 64-bit prowess, September 2004. http://www.linuxhardware.org/article.pl?sid=04/09/17/1453239.

[45] J.-S. 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*, pages 264–274. ACM Press, June 2000.

[46] A. Kleen. Porting linux to x86-64. In *Proceedings of the Linux symposium*, July 2001.

[47] K. Krewell. AMD serves up Opteron. *Microprocessor report*, April 2003.

[48] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Jun 2002.

[49] C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 Workshop on Memory Systems Performance*, pages 24–35. ACM Press, June 2005.

[50] J.-S. Lee, W.-K. Hong, and S.-D. Kim. Design and evaluation of a selective compressed memory system. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, page 184. IEEE Computer Society, November 1999.

[51] O. Lhotk and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 175–184. ACM Press, November 2002.

[52] T. Li, L. K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the 14th International Conference on Supercomputing*, pages 22–33. ACM Press, May 2000.

[53] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[54] Y. Luo and L. John. Workload characterization of multithreaded Java servers. Technical Report Technical Report TR-010815-01, Department of Electrical and Computer Engineering, University of Texas at Austin, June 2001.

[55] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 138–149, June 1995.

[56] F. W. Miller. Simple memory protection for embedded operating system kernels. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 299–308. The USENIX Association, June 2002.

[57] C. Nakhli, C. Rippert, G. Salagnac, and S. Yovine. Efficient region-based memory magement for resource-limited real-time embedded systems. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. Springer Verlag, 2006.

[58] T. Nelson and M. O'Connor. 64-bit computing update. *Processor*, 25(30):editorial article, July 2003.

[59] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, 2001.

[60] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[61] P. Seshadri and A. Mericas. Workload characterization of multithreaded Java servers on two PowerPC processors. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 36–44. IEEE Computer Society, December 2001.

[62] S. T. Shebs and R. R. Kessler. Automatic design and implementation of language data types. In *Proceedings of the ACM SIGPLAN 1987 Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, June 1987.

[63] 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 (POPL)*, pages 295–306. ACM Press, January 2002.

[64] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, June 2001.

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

[66] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. P. Grove, and M. J. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of USENIX 3rd Virtual Machine Research and Technology Symposium(VM'04)*, pages 57–72. The USENIX Association, May 2004.

[67] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *PACT: International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329. IEEE Computer Society, 1998.

[68] K. Venstermans and K. De Bosschere. JVM SPEC favours 32-bit platforms. In *ProRISC*, Veldhoven, the Netherlands, November 2003. http://escher.elis.ugent.be/publ/Edocs/DOC/P103_143.pdf.

[69] K. Venstermans, L. Eeckhout, and K. De Bosschere. 64-bit versus 32-bit virtual machines for Java. *Software—Practice and Experience*, 36(1):1–26, January 2006.

[70] K. Venstermans, L. Eeckhout, and K. De Bosschere. Space-efficient 64-bit Java objects through selective typed virtual addressing. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, pages 76–86. IEEE Computer Society, March 2006.

[71] K. Venstermans, L. Eeckhout, and K. De Bosschere. Java object header elimination for reduced memory consumption in 64-bit

virtual machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2007. To be published.

[72] K. Venstermans, L. Eeckhout, and K. De Bosschere. Object-relative addressing: Compressed pointers in 64-bit Java virtual machines. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science*. Springer, July 2007. To be published.

[73] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265. ACM Press, December 2000.

[74] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *Computational Complexity*, pages 14–28, 2002.

[75] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *SIGPLAN Notices*, 35(11):150–159, 2000.