64-bit versus 32-bit Virtual

Machines for Java

Kris Venstermans, Lieven Eeckhout and Koen De Bosschere

Department of Electronics and Information Systems (ELIS), Ghent University, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium {kvenster, leeckhou, kdb}@elis. UGent. be



SUMMARY

The Java language thanks its popularity to its platform independence, making it useful in a lot of technologies ranging from embedded devices to high-performance systems. The platform independent property of Java, which is visible at the Java bytecode level, is only made possible thanks to the availability of a Virtual Machine (VM) that needs to be designed specifically for each underlying hardware platform. More specifically, the same Java bytecode should run properly on both a 32-bit or a 64-bit VM. In this paper, we compare the behavioral characteristics of 32-bit versus 64-bit virtual machines using a large set of Java benchmarks. This is done using the Jikes Research VM as well as the IBM JDK 1.4.0 production VM on a PowerPC-based IBM machine. By running the PowerPC machine in both 32-bit and 64-bit mode we are able to compare 32-bit versus 64-bit VMs. We conclude that the space an object takes in the heap in 64-bit mode is 39.3% larger on average than in 32-bit mode. We identify three reasons for this: (i) the larger pointer size, (ii) the increased header and (iii) the increased alignment. The minimally required heap size is 51.1% larger on average in 64-bit than in 32-bit mode. From our experimental setup using hardware performance monitors, we observe that 64-bit computing typically results in a significantly larger number of data cache misses at all levels of the memory hierarchy. In addition, we also observe that when a sufficiently large heap is available, the IBM JDK 1.4.0 VM is 1.7% slower on average in 64-bit mode than in 32-bit mode.

KEY WORDS: Java; Virtual Machine; 64-bit vs. 32-bit computing; performance evaluation; PowerPC

INTRODUCTION

Java applications are becoming increasingly more common on various computing technologies, ranging from embedded devices to desktop environments to high-performance systems. This is due to its platform independent nature which is made possible by the availability of a Virtual Machine (VM) for each specific hardware platform. In other words, the same Java bytecode can be executed on different hardware environments provided that a VM is available for the given hardware platform. Nowadays a multitude of VMs exist to support a large variety of instruction set architectures (ISAs) such as IA-32, IA-64, PowerPC, Alpha, MIPS, Sparc, etc. Obviously, Java applications should also run on 32-bit as on 64-bit machines.

In this paper we are specifically interested in 32-bit versus 64-bit Java processing and its impact on performance. This research is motivated by the fact that although the 64-bit world is not new to a variety of high-end servers and applications, its growing popularity towards consumer desktop computers will make it the major universe *tout court* within a few years. The main reason that the market for consumer applications is running behind, is that this market is dominated by IA-32 ISA based hardware. However, AMD recently announced 64-bit hardware extending the IA-32 ISA in its Athlon64 [11] and Opteron [15] microprocessors. Currently, Intel is also working on 64-bit x86 processors, also called Intel Extended Memory 64 Technology (EM64T) [12]. With the availability of 64-bit hardware in the desktop market, soon everyone will be confronted with 64-bit applications. So it is important for application developers, virtual machine designers and computer architects to have a clear understanding of how different 64-bit Java behaves from 32-bit Java.

In order to compare a 32-bit versus a 64-bit Java environment, we use two VMs, namely the Jikes Research VM and the IBM JDK 1.4.0 production VM, on a PowerPC-based IBM POWER4 hardware platform. Both VMs can be run in 32-bit as well as in 64-bit mode on the same hardware which makes this setup an excellent opportunity to investigate the impact

of 32-bit versus 64-bit VMs on memory and overall 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 consumption and the impact on execution speed. To the best of our knowledge, this paper is the first one to investigate the impact of 32-bit versus 64-bit computing in the context of Java workloads. We study and quantify both the increased memory requirements due to 64-bit computing and its impact on overall performance.

We show that the space an object takes in the heap increases by 39.3% on average when using a 64-bit VM versus a 32-bit VM. We identify three reasons for this: (i) the increased pointer size (64 bits versus 32 bits), (ii) the increased header and (iii) the increased number of bytes that need to be inserted into the objects for alignment purposes. We also quantify the increased heap size that is required for Java applications when run in 64-bit mode. To this end, we define the critical heap size, or the minimum heap size for which near optimal performance is achieved. From our experimental setup, we conclude that the critical heap size is 39.5% larger for 64-bit VMs than for 32-bit VMs. Also the minimum heap size for which a Java application still runs without crashing as a result of not having enough memory, increases by 51.1%. Finally, we also quantify the impact on memory and overall performance and study the behavioral characteristics using hardware performance monitors of a 32-bit versus a 64-bit VM. 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 a 9.9%, 29.0% and 38.4%larger number of data cache misses when comparing 64-bit versus 32-bit computing for the L1, L2 and L3 data caches, respectively. Finally, when considering overall performance, we conclude that when a sufficiently large heap is available, Java applications run 1.7% slower on average on a 64-bit VM than on a 32-bit VM.

The observation that 64-bit computing involves increased memory requirements is not surprising. In fact, the larger data footprint for 64-bit versus 32-bit computing is already subject of research addressing it, see for example the work done by Adl-Tabatabai *et al.* [1].

However, as mentioned before there is no prior work with quantitative measurements on the increased memory requirements of 64-bit versus 32-bit computing for Java applications. In fact, we are the first to quantify the increase in object and heap size and its impact on memory and overall performance due to 64-bit computing. As such, we feel that the detailed measurements presented in this paper are useful for future research attacking the problem of the excessive memory consumption in 64-bit computing.

The rest of this paper is organized as follows. We first present our experimental setup and detail on the key differences between 64-bit and 32-bit VMs. We next highlight its impact on object size and we subsequently quantify and compare the memory requirements of 64-bit versus 32-bit VMs and its impact on overall performance. Finally, we discuss related work after which we conclude.

EXPERIMENTAL SETUP

This section describes the experimental setup for this paper. We discuss the virtual machines, the hardware platform and the benchmarks used throughout this paper.

Virtual machines

In this paper we use two virtual machines, the Jikes Research VM (Jikes RVM) and the IBM JDK 1.4.0 production VM (IBM VM). The Jikes RVM [2] is an open-source virtual machine, developed by IBM Research^{*}. 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 recently ported to 64-bit. The Jikes RVM compiles Java bytecode upon its first invocation 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. The

^{*}http://www.ibm.com/developerworks/oss/jikesrvm

Jikes RVM is itself written in Java. In this paper, we will use 3 garbage collectors (GCs) for Jikes RVM [5], namely SemiSpace which is the simplest copying GC (worst heap consumption), MarkSweep which is the simplest non-copying GC (worst collector-specific overhead), and GenMS (generational MarkSweep), which is the best performing GC of Jikes RVM and which is in essence a combination of a copying collector (as in SemiSpace) and MarkSweep. Assertion checking was disabled in all measurements using the Jikes RVM. Recently, Jikes RVM was extended to support reading the Hardware Performance Monitors per thread [22]. We make use of this extension for the extraction of GC specific information via the GC thread. In our measurements we use release 2.3.2, patched with code for a better inter-object alignment strategy[†]. This patch will be integrated in the future releases of Jikes RVM, but was not yet available in the 2.3.2 release.

The IBM JDK 1.4.0[‡] VM [21] uses a mixed scheme of interpretation and compilation. It is a production VM, meaning that it highly optimizes during compilation. The user has the choice between throughput optimization or pause time optimization. We use both variants in our measurements. The garbage collection strategy of the IBM JDK 1.4.0 VM is based on a Mark-Sweep-Compact strategy.

We used these two virtual machines for the following reasons. First, the Jikes RVM has the important advantage over other VMs of being open-source which allows us to add instrumentation code to the VM to measure information of interest. This will be done in this paper to measure for example object size. Second, since porting the Jikes RVM from 32bit mode to 64-bit mode for PowerPC was done very recently (as of release 2.3.2, April 2004), the current 64-bit mode only supports the baseline compiler, i.e. the optimizing compiler is not supported yet. Because comparing the 64-bit versus the 32-bit Jikes' baseline compiler would not have been representative for real environments, we decided to include a production compiler as well, namely the IBM JDK 1.4.0 VM. Both virtual machines can be run in 32-

[†]http://www.elis.ugent.be/~kvenster/patchAlignmentJikes232 [‡]http://www.ibm.com



cache	size	line size	associativity
L1 I-cache	64KiB	128B	direct mapped
L1 D-cache	32 KiB	128B	2-way set assoc.
L2 unified	$1.41 \mathrm{MiB}$	128B	8-way set assoc.
L3 unified	32 MiB	512B	8-way set assoc.

Table I. Cache hierarchy of the IBM POWER4.

bit and 64-bit mode. We believe that for the purpose of quantifying object sizes and related issues, using the baseline Jikes RVM does not impact the overall conclusions. For reporting performance numbers, we use the production IBM JDK 1.4.0 VM.

PowerPC platform

The IBM POWER4 [3], on which the hardware measurements in this paper are done, is a 64bit 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 machine has 1GiB[§] of RAM. 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 L1 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. It 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 I.

[§]KiB, MiB, etc.: see http://physics.nist.gov/cuu/Units/binary.html

Copyright © 2005 John Wiley & Sons, Ltd. Prepared using speauth.cls

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 pages of 4KiB but also supports large 16MiB pages. In our measurements we only use 4KiB pages for both the 32-bit and 64-bit VMs. 16MiB 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 of this paper we will characterize the behavior of the 32-bit and 64-bit VMs 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 (pmapi library) 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.

Benchmarks

The benchmark set was constructed from a variety of sources: SPECjvm98, SPECjbb2000, Java Grande Forum suite and Xalan. Each benchmark is given a short description in Table II. SPECjvm98[¶] is a client-side Java benchmark suite, for which we used the default, s100 input set. Pseudojbb 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. A

[¶]http://www.spec.org/jvm98 |http://www.spec.org/jbb2000



	1 able 11. Benchmarks used in our experimental setup.
Xalan	XSLT processor for transforming XML documents into HTML, text, or
	other XML document types.
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.
SPECjvm98	
202_{jess}	An expert shell system, based on NASA's CLIPS expert system, solving
	a set of puzzles with varying degree of difficulty.
209_db	This benchmark makes some database requests on a memory resident
	database.
213_javac	The JDK 1.0.2 Java to bytecode compiler.
222_mpegaudio	A commercial application decompressing MPEG-3 audio files. This
	benchmark makes very few allocation requests.
227_mtrt	A dual-threaded raytracer program.
228_jack	An early version of the JavaCC Java source code parser generator.
Java Grande H	orum
search	A program solving a connect-4 game, using an alpha-beta pruning
	technique. N positions are evaluated. $(N = 34, 517, 760)$
moldyn	Evaluation of an N-body model for particles interacting under a Lennard-
	Jones potential in a cubic space. $(N = 8, 788)$
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. $(N = 2,000)$
SOR	This benchmark performs 100 iteration of successive over-relaxation on
	a N x N grid. $(N = 2,000)$
sparseMatMult	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 handbranks exercises indirection addressing and non
	structure. This benchmarks exercises indirection addressing and non-

Table II. Benchmarks used in our experimental setup.

very different set of applications are the sequential benchmarks from the Java Grande Forum suite^{**}. 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. Finally, we also use Xalan^{††}, an XSLT processor for XML parsing.

32-BIT VERSUS 64-BIT VM

In this section we will highlight some key differences between the 32-bit and 64-bit versions of the Jikes RVM. Unfortunately, we are unable to verify whether the same differences exist between the 32-bit and the 64-bit versions of the IBM VM since the source code of the latter is not available. However, we believe that the differences described below are general enough to assume that they will also exist in the IBM VM. We identify four key differences: (i) the extended ISA, (ii) the increased object size due to larger pointers and alignment, (iii) the increased stack size, and (iv) argument passing.

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 machines would require a sequence of instructions to compute the same result. 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.

^{**}http://www.javagrande.org

^{††}http://xml.apache.org/xalan-j/index.html



Increased object size

There are three reasons why objects as they are allocated in the heap are larger in a 64-bit VM than in a 32-bit VM. The first reason obviously is the increased size of the pointers, namely 64 bits instead of 32 bits. The second reason concerns the header. 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[†]. However, we observed there is room for improvement because 64-bit Jikes does not fully use all words at all times, e.g. array length needs only 4 bytes, 4 out of 8 bytes in the status word are only used for copying GCs, and the Type Information Block (TIB) word could possibly be compressed. The third reason for the increased object size 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 on 4-byte boundaries is sufficient. Note that 64-bit fields (in both 32-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 paper, we will refer to this padding as intra-object alignment. In general, all previous reasons cause the object size to increase when used in 64-bit mode and they will potentially impact e.g. cache and TLB behavior.

Increased stack size

The sizes for the different Java types when used as object fields in Jikes RVM, i.e. when allocated in the heap, are listed in Table III 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 de-allocated. As shown in the 'size on stack' column in

[†]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.

	32-bit	platform	64-bit platform		
	Field	Size on	Field	Size on	
Java types	size	stack	size	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	

Table III. Java types a	and their sizes in m	umber of bits when	used in the heap ('field
size' column)	and when used on	the stack ('size on	stack' column).

Table III, 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 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 excessively 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 paper. We refer the interested reader to [23] for more details.

Argument passing

Conventions for argument passing may also cause differences between 32-bit and 64-bit VMs. In Jikes RVM's baseline compiler, 13 floating point registers and 8 general purpose registers can be used for argument passing. Passing a long in 32-bit mode requires 2 general purpose registers, but requires only 1 register in 64-bit mode, leaving more registers available for other arguments/purposes.



		overall		8	rray obje	cts	non-array objects		
Benchmark	32-bit	64-bit	increase	32-bit	64-bit	increase	32-bit	64-bit	increase
Xalan	106.1	134.5	26.7%	208.5	250.7	20.3%	24.3	41.7	71.4%
JGFCrypt	3124.0	3238.4	3.7%	7563.2	7818.1	3.4%	25.9	42.7	64.9%
JGFHeapSort	2152.5	2267.9	5.4%	5201.4	5458.4	4.9%	25.9	42.7	64.8%
JGFLUFact	760.7	866.5	13.9%	1724.0	1947.3	13.0%	25.9	42.7	64.8%
JGFMolDyn	134.6	219.6	63.1%	306.9	513.3	67.3%	35.8	51.1	42.7%
JGFSOR	767.3	880.2	14.7%	1737.1	1975.6	13.7%	25.9	42.6	64.8%
JGFSearch	44.3	52.5	18.4%	44.3	52.5	18.4%	25.9	42.6	65.0%
JGFSparseMatmult	1110.5	1226.8	10.5%	2665.5	2924.5	9.7%	25.8	42.6	64.9%
_202_jess	35.8	59.0	65.0%	52.4	97.7	86.6%	27.4	39.5	44.2%
_209_db	26.7	50.3	88.5%	202.2	360.4	78.3%	16.5	32.3	95.9%
_213_javac	35.0	53.0	51.7%	58.1	78.7	35.4%	24.4	41.3	69.4%
_222_mpegaudio	111.1	172.1	55.0%	233.4	356.6	52.8%	24.1	40.9	69.7%
_227_mtrt	23.3	33.9	45.6%	34.5	50.7	47.1%	20.4	29.5	44.9%
_228_jack	39.1	56.1	43.7%	53.0	71.6	35.1%	25.5	41.0	61.1%
pseudojbb	32.8	47.7	45.1%	42.3	55.9	32.2%	27.0	42.5	57.6%
avg			36.7%			34.6%			63.1%

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

Table V. Average	e object size	(in bytes)	in 32-bit	and 64-bit	VM	mode f	for all	objects,	array	objects
	and n	on-array o	bjects in	the default	obje	ct space	e(s).			

	overall			array objects			non-array objects		
Benchmark	32-bit	64-bit	increase	32-bit	64-bit	increase	32-bit	64-bit	increase
Xalan	67.8	85.1	25.6%	122.4	139.8	14.3%	24.2	41.6	71.6%
JGFCrypt	55.7	71.8	29.0%	98.6	113.9	15.5%	25.8	42.6	65.0%
JGFHeapSort	55.2	71.3	29.3%	97.3	112.6	15.8%	25.8	42.6	65.0%
JGFLUFact	55.7	71.4	28.4%	98.6	113.0	14.6%	25.8	42.6	64.9%
JGFMolDyn	59.6	75.0	25.8%	101.1	116.7	15.3%	35.8	51.1	42.8%
JGFSOR	55.1	71.2	29.1%	97.3	112.3	15.4%	25.8	42.6	65.0%
JGFSearch	44.0	52.0	18.2%	44.0	52.0	18.2%	25.8	42.6	65.1%
JGFSparseMatmult	55.0	71.1	29.5%	96.8	112.1	15.8%	25.8	42.5	65.1%
_202_jess	35.0	57.6	64.6%	50.1	93.5	86.6%	27.4	39.5	44.2%
_209_db	18.1	33.8	86.3%	46.6	59.4	27.5%	16.5	32.3	95.9%
_213_javac	33.9	50.9	50.0%	54.8	71.9	31.2%	24.4	41.3	69.4%
_222_mpegaudio	59.4	75.8	27.6%	109.2	125.0	14.4%	24.0	40.8	69.9%
_227_mtrt	22.4	32.2	44.0%	29.9	42.5	41.9%	20.4	29.5	44.9%
_228_jack	38.2	54.6	43.0%	51.3	68.6	33.8%	25.5	41.0	61.1%
pseudojbb	32.7	47.4	45.1%	41.8	55.2	32.0%	27.0	42.5	57.6%
avg			38.4%			26.2%			63.2%

ALLOCATION BEHAVIOR

In this section, we quantify the allocation behavior and the increase in memory requirements when comparing the 64-bit VM versus the 32-bit VM. We first measure the increase in average object size. We subsequently discuss how this affects the object size distribution. And finally, we quantify the increase in heap size.

Average object size

As pointed out in the previous section, the object size increases when comparing 64-bit versus 32-bit because of three reasons: (i) 64-bit versus 32-bit pointers, (ii) the header doubling in size, and (iii) additional padding for intra-object alignment. Before presenting our measurements on the object size, it is interesting to make the following comment. Since the Jikes RVM is itself written in Java, our measurements include application objects as well as objects supporting the internals of the RVM. 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, see for example [8]. As a result of that, the average object sizes reported in those papers typically are smaller than the ones presented in this paper. This difference is due to the fact that a VM typically allocates large structures, e.g. stacks, which increase the average object size. This relates to all data presented in this paper.

Table IV presents the average object size in 32-bit and 64-bit mode along with its relative increase. This is done for all objects, array objects and non-array objects. For non-array objects, we observe an increase in size from 32-bit to 64-bit that is nearly constant over all benchmarks. The average increase is around 16 bytes. Recall from the previous section that for non-array objects 8 of these 16 bytes come from the increased header. The remaining increase thus comes from alignment and larger pointers in the fields of the object—this suggest that a non-array object has one or two references on average among its fields. The average relative increase in size for non-array objects is 63.1%. For array objects, the picture is different (34.6% on average):

some benchmarks have small object size increases (3.4% for JGFCrypt) whereas others suffer from large object size increases (86.6% for jess). This suggests that most arrays in jess contain references (which all double in size) whereas for JGFCrypt, most arrays do not contain references (those only have a header increase). When considering all objects (both array and non-array objects), we observe an average object size increase of 36.7%. The average increase is larger for SPECjvm98 (58.2%) than for Java Grande Forum (18.5%). It is also interesting to note that the Java Grande Forum benchmarks typically have larger objects than the SPECjvm98 benchmarks. This is due to the frequent use of large numeric arrays in the JGF benchmarks.

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 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 for exploiting spatial and temporal locality—for example, putting objects that are related to each other on the same cache line [7]. 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 since it maintains a Large Object Space and an Immortal Space next to (a) collector-specific space(s) (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 KiB for the MarkSweep GC and the GenMS GC and 16 KiB for the SemiSpace collector. The Immortal Space contains objects that should not be collected, i.e. specific RVM objects. Table V 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 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 IV. This is due to the array objects since for non-array objects, there is no significant difference between Tables IV and V. Array objects that are allocated in the default space show a nearly constant increase in the object size when comparing 32-bit versus

64-bit mode, namely 16.1 bytes on average. Of these, 12 bytes will go to the (aligned) header (see previous section); the remainder goes to alignment and references—by consequence, this suggests the number of reference arrays in this space is small and/or the reference arrays are very small in size. In general, the average object size increases by 38.4%. In terms of cache lines, an IBM POWER4 L1 D-cache line can hold 2.8 and 2.1 objects on average in 32-bit and 64-bit mode, respectively.

Object size distribution

In the previous section we distinguished objects in the default space versus objects in the Large Object Space and the Immortal Space to get an estimate of small versus large objects. We now go one step further by studying the object size distribution. This will allow us to study the impact of 64-bit versus 32-bit mode for different object sizes. Figures 1 and 2 present two example object size distributions for jess and pseudojbb, respectively—similar results were obtained for the other benchmarks. Within each figure, the three left graphs show the object size distribution for the 32-bit mode; the three right graphs are for the 64-bit mode. The top graphs in each figure show the object size distribution for all objects (array plus non-array objects); the middle graphs are for array objects and the bottom graphs are for non-array objects. We observe that non-array objects are typically smaller than 100 bytes, even in 64-bit mode. Arrays have a more widespread size distribution: the tail of their distribution is heavier than for non-array objects. For some benchmarks, the object size distribution shows peaks for larger object sizes, e.g. pseudojbb has a fairly large amount of arrays with sizes between 600 and 1,000 bytes, see Figure 2. For several Java Grande Forum benchmarks, we observed arrays of several thousands of bytes. If we compare the 32-bit mode versus the 64-bit mode object size distributions, we conclude that, as expected, the distribution shifts towards larger object sizes. This is particularly true for the non-array objects. In addition, the distribution seems to change its shape. This is because not all objects increase in size by a fixed number of bytes; some objects double in size whereas others only increase incrementally. Note that the





Figure 1. Object size distribution for jess, from top to bottom: all objects, array objects and non-array objects. The left graphs are for 32-bit mode, the right graphs are for 64-bit mode. The horizontal axes are shown on a log scale for the top graphs and on a linear scale for the middle and bottom graphs.

increased object size does not necessarily needs to be a disadvantage for 64-bit mode versus 32-bit mode. For example, consider the case that the object size in 32-bit mode is smaller than the Large Object Space threshold and the object size in 64-bit mode is larger than the Large Object Space threshold. In that case the object in 64-bit mode will be allocated in the Large Object Space whereas the object will be allocated in the default space in 32-bit mode. The





Figure 2. Object size distribution for pseudojbb, from top to bottom: all objects, array objects and non-array objects. The left graphs are for 32-bit mode, the right graphs are for 64-bit mode. The horizontal axes are shown on a log scale for the top graphs and on a linear scale for the middle and bottom graphs.

potential benefit for an object being allocated in the Large Object Space is that it will not be copied by the garbage collector. Obviously, this is only an advantage in case of a copying collector; a non-copying collector does not copy objects at all. From our measurements, we conclude however that this effect is marginal: these objects represent no more than 0.06% of all bytes allocated.



		32-bit			64-bit		
	object	+inter-object		object	+inter-object		1
Benchmark	size	alignment	in heap	size	alignment	in heap	increase
Xalan	106.2	106.2	120.6	134.5	137.7	152.0	26.1%
JGFCrypt	3124.0	3124.2	3134.4	3238.4	3241.4	3251.5	3.7%
JGFHeapSort	2152.5	2152.7	2162.7	2267.9	2270.8	2280.7	5.5%
JGFLUFact	760.7	761.1	784.3	866.5	869.5	892.2	13.8%
JGFMolDyn	134.6	135.3	146.2	219.6	222.2	231.7	58.5%
JGFSOR	767.3	767.7	791.4	880.2	883.1	906.4	14.5%
JGFSearch	44.3	44.3	44.9	52.5	56.5	57.4	27.9%
JGFSparseMatmult	1110.5	1110.7	1120.7	1226.8	1229.8	1239.8	10.6%
_202_jess	35.8	36.4	37.4	59.0	61.7	63.9	70.9%
_209_db	26.7	26.7	27.5	50.3	50.9	51.7	88.0%
_213_javac	35.0	35.0	37.0	53.0	55.8	57.9	56.5%
_222_mpegaudio	111.1	111.2	121.7	172.1	175.2	185.5	52.4%
_227_mtrt	23.3	23.3	23.9	33.9	38.3	39.1	63.7%
_228_jack	39.1	39.2	43.1	56.1	60.6	62.5	45.1%
pseudojbb	32.8	33.3	35.2	47.7	51.1	53.6	52.1%
avg							39.3%

Table VI. Object size, increase due to inter-object alignment and object heap size in 32-bit and 64-bit mode for the MarkSweep collector. The right most columns shows the average increase in heap size when going from 32-bit mode to 64-bit mode.

Heap growth

Even more interesting when evaluating a VM, is not the object size *per se* as done in the previous subsections, but the number of bytes allocated for an object in the heap which we will call the *object heap size*. This metric is very collector (and implementation) specific. The object heap size is generally larger than the object size. A first source of overhead is interobject alignment—not to be confused with the intra-object alignment as discussed in previous sections. 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. A second source of overhead comes from the memory manager. The memory manager can result in different kinds of overhead. A memory manager typically keeps track of the allocated objects in memory blocks (a number of contiguous memory pages); this information is maintained in a data structure on the given memory block. Also, the memory manager might not be able to completely fill up a memory block with allocated objects due to the fact that the number of





Figure 3. Heap growth for MarkSweep collector as a function of time (measured per allocation site).

available bytes is smaller than the number of bytes needed for allocating new objects (external fragmentation). Finally, some memory managers use fixed-sized cells for allocating objects. When the object size does not match the cell size used, an additional overhead incurs, i.e. a number of bytes remain unused in a cell (internal fragmentation). Note that the additional overhead due to the memory manager is very much dependent on the given memory manager.

To quantify the extra heap overhead, we first consider the MarkSweep collector in Jikes RVM because it is the collector with the worst overhead of all 3 collectors used. The MarkSweep collector in Jikes RVM uses cells per page and a data structure to keep track of used and unused cells per page. The results are shown in Table VI: the object size, the additional overhead due to inter-object alignment and the additional overhead due to the memory manager (the object heap size). The average increase in heap size when going from 32-bit mode to 64-bit mode is 39.3%, which is larger than the increase in object size (36.7% see Table IV), this is due to





Figure 4. Heap growth for SemiSpace collector as a function of time (measured per allocation site).

the bigger alignment cost in 64-bit mode. During our analysis of the additional overhead due to inter-object alignment, we observed that the memory allocator in 64-bit Jikes RVM could be improved. Our improved implementation (see also the section on our experimental setup for more details) reduced this inter-object alignment from 9.9 bytes to 3.1 bytes on average in 64-bit mode—we used the improved memory allocator in our measurements. The extra overhead of the memory manager is almost the same number of bytes when comparing 64-bit versus 32-bit. The overhead of the memory manager is 4.2% and 2.7% in 32-bit and 64-bit mode, respectively. As mentioned above, the MarkSweep collector shows the worst overhead of all collectors considered here. The SemiSpace collector shows the lowest memory manager overhead: only 1.1% and 0.9% in 32-bit and 64-bit mode, respectively, which is significantly smaller than for the MarkSweep collector. For the generational GenMS collector, the extra overhead is not constant over time, because the generational collector copies objects from the





Figure 5. Heap growth for GenMS collector as a function of time (measured per allocation site).

nursery to the mature space. In each of the spaces a different collector is used: the nursery uses the SemiSpace collection strategy whereas the mature space uses MarkSweep. Objects in the nursery will thus experience the low overhead of the SemiSpace collector; objects in the mature space will experience the higher overhead of the MarkSweep collector. As such, the average overhead of the GenMS collector lies somewhere between the SemiSpace and MarkSweep collectors.

Table VI provided the average size per allocated object, i.e. the size for each allocated object is used to compute the average object heap size. However, the actual heap only contains live objects (or more correctly, the objects not discovered to be dead yet). By consequence, the actual heap growth when comparing 64-bit vs. 32-bit computing might be different than the average 39.3% object heap size 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 the previous time, 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 Figures 3, 4 and 5, the heap size is shown as a function of time measured by the number of allocations for the MarkSweep, the SemiSpace and the GenMS collector, respectively. This is done for four benchmarks, search, pseudojbb, mtrt and jack. In all these experiments the maximum heap size was set large enough so that the actual heap size never reaches this maximum. As expected, the heap size in 64-bit mode never reaches twice the 32-bit mode heap size, however, in several cases, for example for jack and mtrt, we observe that the 64-bit VM uses nearly twice as much heap size as the 32-bit VM.

For the SemiSpace and the MarkSweep collectors, we observe that the 32- and 64-bit VMs typically exhibit similar growing/shrinking behavior, except for the beginning, where the 64-bit version needs 1 or 2 extra GCs. These extra GCs could be easily avoided by simply setting the initial heap size twice the value in 32-bit mode. We did not set different initial heap sizes for 32- and 64-bit mode though. The heap behavior for GenMS collector is slightly different. We can draw the same conclusion if we only take into account the mature space collections. The GenMS collector does trigger a large number of collections in the nursery without adjusting the heap threshold. Only after a mature space collection is triggered, the heap threshold can increase. This leads to a much higher number of nursery collections in 64-bit mode.

We now quantify the *critical heap size*. The critical heap size is defined as the minimum heap size for which the given benchmark reaches the same execution time as an almost infinite maximum heap size, i.e. no additional GCs are needed compared to an infinite maximum heap size. We computed the critical heap size by doing multiple measurements for different maximum heap sizes. Then we determine the critical heap size H_c for which we do not observe more than 2% extra GC-time compared to an infinite maximum heap size. In some cases a value higher than H_c occasionally takes more than 2% extra GC-time. We did not take into account these spurious values when determining H_c . As an example, Figure 6 shows the

	32	32 bit		bit
Benchmark	H_m	H_c	H_m	H_c
_202_jess	12	61	22	113
_209_db	24	51	44	104
_213_javac	26	76	49	118
_222_mpegaudio	9	17	17	21
_227_mtrt	22	78	44	159
_228_jack	11	68	19	119
Xalan	19	42	31	57
JGFCrypt	150	150	156	156
JGFHeapSort	102	102	108	109
JGFLUFact	68	68	74	74
JGFSOR	68	68	74	74
JGFSparseMatMult	54	54	60	60
JGFSearch	12	101	18	160
JGFMoldyn	9	14	15	18
pseudojbb	289	692	358	662

Table VII. Minimal heap size (H_m) and critical heap size (H_c) measured in MiB for the SemiSpace collector.

	32 bit		64	bit
Benchmark	H_m	H_c	H_m	H_c
_202_jess	10	20	17	28
_209_db	16	24	29	47
_213_javac	18	23	34	66
_222_mpegaudio	8	17	14	20
_227_mtrt	14	20	28	31
_228_jack	9	20	16	27
Xalan	16	25	23	35
JGFCrypt	149	149	155	155
JGFHeapSort	102	102	107	107
JGFLUFact	37	37	43	43
JGFSOR	37	37	43	43
JGFSparseMatMult	52	52	58	58
JGFSearch	11	20	17	20
JGFMoldyn	7	11	13	17
pseudojbb	171	494	244	501

Table VIII. Minimal heap size (H_m) and critical heap size (H_c) measured in MiB for the GenMS collector.



Figure 6. Defining the critical heap size H_c for jess and pseudojbb in 64-bit mode. The execution time spent in GC is shown as a function of the heap size.

execution time as a function of the maximum heap size along with the critical heap size H_c for two benchmarks jess and pseudojbb[‡]on 64-bit. For all the benchmarks, we computed the critical heap size H_c for the SemiSpace collector (worst heap usage) as well as for the GenMS collector (best heap usage), see Tables VII and VIII. The critical heap size increases by 39.5% on average when comparing 64- versus 32-bit computing. Tables VII and VIII also show the minimal heap size H_m per benchmark. The minimal heap size is defined as the heap size for which the given benchmark does not crash because of not having enough memory. We observe that the minimal heap size H_m increases by 51.1% on average when comparing 64-bit versus 32-bit mode. It is interesting to note that the minimal heap size increases more than the critical heap size does when comparing 64-bit versus 32-bit. We can conclude from this that especially long-lived objects will increase in size when going from 32-bit mode to 64-bit mode.

Obviously, running a benchmark with a maximum heap size that is lower than the critical heap size, results in an increase in garbage collections. In some cases, the overall execution time might even get dominated by the time spent during collection. Figure 7 shows the heap

[‡] In contrast to the other benchmarks, H_c for pseudojbb is larger on 32-bit than on 64-bit. This is because we only have 1GiB of RAM and for bigger heap sizes we noticed more swapping. As a result H_c is pushed more towards a smaller value in 64-bit mode. The real H_c for systems with more RAM available will be higher.





Figure 7. Heap growth for jack as a function of time (measured per allocation site). Maximum heap size is set to 68MiB, the critical heap size in 32-bit mode.

behavior of jack when run on 32- and 64-bit VMs with the SemiSpace collector; in these experiments the maximum heap size was set to the critical heap size under 32-bit mode, i.e. 68MiB. Under 64-bit mode VM we measure 22 collections. This is significantly more than the 16 collections that occur when running under the 64-bit critical heap size.

OVERALL PERFORMANCE

In this section, we quantify the impact on execution time when comparing 64-bit versus 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 IBM production VM under the two optimization schemes, maximum throughput and minimum average pause time.

Execution time

Figure 8 shows the ratio in execution time for 64-bit mode compared to 32-bit mode. The left graph is for the maximum throughput optimization scheme; the graph on the right for the minimum pause time scheme. Values higher than 1 thus indicate 64-bit mode is slower than 32-

26 K. VENSTERMANS, L. EECKHOUT, K. DE BOSSCHERE





Figure 8. Ratio in execution time between 64-bit and 32-bit mode.



Figure 9. Ratio in the number of executed instructions between 64-bit and 32-bit mode.

bit mode execution. In general, 64-bit mode is slower than 32-bit mode. For some benchmarks, we observe performance decreases up to 20% (jess under maximum throughput), 28% (db under maximum throughput), 35% (db under minimum pause time) and 47% (JGFCrypt). For other benchmarks, 64-bit is faster than 32-bit computing, for example JGFSearch (40%), xalan (35%) and jack (32%). However, nearly half the benchmarks are unaffected by 64- vs. 32-bit computing or experience a small performance decrease. On average, the IBM VM is 1.7% slower in 64-bit mode than in 32-bit mode, mainly due to the larger memory footprint (see the next subsections about data memory performance). In the following subsections we will provide explanations for the observed behavior. The mean values of ratios are calculated as a geometric mean.

Copyright © 2005 John Wiley & Sons, Ltd. Prepared using speauth.cls Softw. Pract. Exper. 2005; 00:1-33



Figure 10. Ratio in the number of L1 D-cache misses between 64-bit and 32-bit mode.

Number of instructions executed

SP&E

Figure 9 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. On average however, we observe a decrease of 5.0%. Four benchmarks, namely xalan, JGFSearch, JGFCrypt and jack are behaving differently from the rest. These four benchmarks have a significantly lower dynamic instruction count under 64-bit mode than under 32-bit mode. From a detailed analysis we discovered that JGFCrypt and JGFSearch perform a large number of arithmetic operations on longs. As such, they can benefit from the 64-bit instructions available in 64-bit mode. For the other two benchmarks, xalan and jack, the lower dynamic instruction count is only observed for the pause time optimization scheme; for the maximum throughput optimization scheme, the dynamic instruction count is nearly equal under 32- and 64-bit computing. This suggests that the pause time optimization scheme executes significantly less instructions in the garbage collector for these benchmarks. The significantly smaller dynamic instruction count explains the better performance under 64-bit mode for JGFSearch, xalan and jack. For JGFCrypt on the other hand, we measured a higher number of branch mis-predictions and more instruction cache misses in 64-bit mode, so that the smaller dynamic instruction count does not reflect itself in better performance.

28 K. VENSTERMANS, L. EECKHOUT, K. DE BOSSCHERE





Figure 11. Ratio in the number of L2 D-cache misses between 64-bit and 32-bit mode.



Figure 12. Ratio in the number of L3 D-cache misses between 64-bit and 32-bit mode.

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 D-TLB behavior. Figures 10, 11 and 12 show the ratio between 64-bit mode and 32-bit mode in the number of D-cache misses at the L1, L2 and L3 level, respectively. A value greater than 1 indicates that 64-bit mode results in an increased number of D-cache misses compared to 32-bit mode. Note that we use the number of misses and not (the commonly used) miss rate or misses per instruction as our metric. This is because miss rate and misses per instruction are related to the number of memory accesses and the number of instructions, respectively, and these denominators are not equal under 32-bit and 64-bit mode. Given the increased object size, the increased alignment, and by consequence an increased heap size, we expect that 64-





Figure 13. Ratio in the number of D-TLB misses between 64-bit and 32-bit mode.

bit VMs will have an increased number of D-cache misses over 32-bit VMs. Figures 10, 11 and 12 show that most benchmarks indeed experience an increase in the number of misses. On average the increase is 9.9%, 29.0% and 38.4% for the L1, L2 and L3 caches, respectively. Note that most Java Grande Forum benchmarks have nearly the same number of misses in 64-bit mode as in 32-bit mode on all cache levels. The SPECjvm98 benchmarks on the other hand, generally have more cache misses in 64-bit mode (see e.g. at the L2 cache level with increases ranging from 20% to 67%). 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 32bit mode (see Table IV). 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 4 in the number of misses. This can be explained in part by the fact that db experiences the highest heap object size increase (88.5%) as reported in Table IV. This high increase in data cache misses explains the higher than 30% performance degradation for db as observed when comparing 64-bit versus 32-bit computing.

D-TLB performance

Figure 13 shows the ratio of D-TLB misses in 64-bit mode versus 32-bit mode. Again, a value greater than 1 denotes an increase in the number of D-TLB misses in 64-bit mode over 32-bit mode. Due to the larger object sizes in 64-bit mode, we expect more pages will get accessed,



and thus we would expect more TLB misses due to an expected increase in the number of conflicts. We observe that the number of D-TLB misses for the IBM VM remains constant for most benchmarks. For several benchmarks we observe a decrease, for example xalan, JGFsearch, mpegaudio and mtrt; for other benchmarks we observe an increase, for example JGFCrypt, jess, db and pseudojbb. On average we observe a(n) (unexpected) decrease in DTLB-misses of 13.8% in 64-mode compared to 32-bit mode. For the throughput optimized VM the decrease is only 6.2% on average, while the minimum pause time optimized VM has a decrease of 20.8% on average.

RELATED WORK

As we stated in the introduction, to the best of our knowledge there is no prior work on comparing 64-bit versus 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 of them (if not all) are done on a 32-bit platform.

Characterizing memory behavior

The study that is probably most related to this work, is the work done by Dieckmann and Hölzle [8] 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 in the heap. And they also compute object size and object alignment.

Shuf et al. [20] 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. [4] present a detailed study on the performance impact of garbage collection. For this use they use hardware performance monitors on three different platforms.

Li et al. [16] 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 [14] 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 [22], method-level phase behavior [10], impact of VMs and input sets on overall Java performance [9], Java middleware benchmarks (SPECjbb2000 and SPECjAppServer2001) using real hardware as well as full-system simulation [13], Java TPC-W which exercises the web server and transaction processing of a typical e-commerce web site [6], cache behavior of SPECjvm98 benchmarks [18], Java server applications (SPECjbb2000 and VolanoMark 2.1.2) on PowerPC hardware [19].

64-bit versus 32-bit pointers

As mentioned in the introduction, the increased memory consumption caused by using 64-bit pointers is not unexpected and has already been subject of research addressing it. Again, we want to emphasize that this previous work did not provide a detailed characterization of the increased memory requirements of 64-bit versus 32-bit computing; this paper is the first to provide such a detailed analysis. As such, we believe the results presented in this paper will be useful in future research attacking the increased memory consumption of 64-bit computing.

Adl-Tabatabai *et al.* [1] study software techniques for reducing 64-bit pointers into 32-bit pointers in the context of a virtual machine running Java applications. Applications that do not need the full 64-bit address space can obviously benefit from this optimization. They study

compressing the object headers and the heap references in memory. They report significant performance improvements for the SPECjvm98 benchmarks, up to 68% (for db)—note db was shown in this paper to have the largest increase in heap object size due the transition from 32-bit to 64-bit computing—as well as for SPECjbb2000 (12%).

Mogul *et al.* [17] study the impact of the pointer size on overall performance. To this end, they consider a number of applications that can run in a 32-bit address space and compare the performance running in 64- vs. 32-bit mode. These measurements were done on a Digital Alpha systems using a collection of C programs. They conclude that while performance was often unaffected by larger pointers, some programs experience definite performance degradation, primarily due to cache and paging issues. Our results are consistent with the observations by Mogul *et al.* although we consider a different programming paradigm; we consider Java workloads opposed to the C programs considered by Mogul *et al.*

CONCLUSION

The purpose of this paper was to compare 64-bit versus 32-bit VMs for Java applications in general and the allocation behavior and memory system performance more in particular. This was done using a large number of benchmarks from SPECjvm98 and Java Grande Forum on two virtual machines, the Jikes Research VM and the production IBM JDK 1.4.0 VM. The underlying hardware platform was the 64-bit PowerPC-based IBM POWER4 processor. By running both virtual machines in 32-bit and 64-bit mode, we were able to compare the characteristics and performance of 32-bit versus 64-bit virtual machines for Java. Using instrumentation inside Jikes RVM, we measured and compared the average object size, the object size distribution and the heap growth for 64-bit and 32-bit computing. We believe that especially VM developers can benefit from the detailed study provided in this paper. We conclude that the average size an object takes in the heap increases by 39.3% in 64-bit mode. This is due to the increased pointer size (64 bits versus 32 bits), the increased header and the increased alignment. The critical heap size, which is the minimum heap size to attain

nearly optimal performance, increases by 39.5%; the minimal heap size for which the VM still runs (at the cost of a larger number of collections), increases by 51.1%. Using the hardware performance monitors available on the POWER4 microprocessor, we were also able to measure the total execution time, the number of instructions executed, the number of data cache and TLB misses for the IBM production VM. We conclude that 64-bit computing is generally slower than 32-bit computing, 1.7% on average. We also conclude that 64-bit Java results in a larger number of data cache misses at all levels in the cache hierarchy which is to be expected given the increased heap size.

ACKNOWLEDGEMENTS

Kris Venstermans is supported by a BOF grant from Ghent University. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research–Flanders (Belgium) (F.W.O.–Vlaanderen). This research is supported by the HiPEAC Network of Excellence. This research is also supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

REFERENCES

- Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Michal Cierniak, Marsha Eng, Jesse Fang, Brian T. Lewis, Brian R. Murphy, and James M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the international symposium on Code Generation and Optimization*, page 100. IEEE Computer Society, March 2004.
- 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.
- Steve Behling, Ron Bell, Peter Farell, Holger Holthoff, Frank O'Connel, and Will Weir. The POWER4 Processor Introduction and Tuning Guide. Redbooks. IBM Corporation, International Technical Support Organization, 2001.
- 4. Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM Press, June 2004.



- Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley". Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society, May 2004.
- Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko 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.
- Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cacheconscious data placement. In *Proceedings of the first International Symposium on Memory Management*, pages 37–48. ACM Press, October 1998.
- Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the specjvm98 Java benchmarks. In 13th European Conference for Object-Oriented Programming, pages 92–115, June 1999.
- 9. Lieven Eeckhout, Andy Georges, and Koen 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, pages 169–186. ACM, October 2003.
- Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen 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, pages 270–287. ACM, October 2004.
- 11. Peter N. Glaskowsky. Athlon 64 moving to mass market. Microprocessor report, January 2004.
- 12. Tom R. Halfhill. AMD and Intel harmonize on 64. Microprocessor report, March 2004.
- Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Javabased middleware. In Proceedings of the Ninth IEEE International Symposium on High-Performance Computer Architecture, pages 217–228. IEEE Computer Society, February 2003.
- 14. Jin-Soo Kim and Yarsun 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.
- 15. Kevin Krewell. AMD serves up Opteron. Microprocessor report, April 2003.
- 16. Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama 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.
- Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava. Performance implications of multiple pointer sizes. In USENIX Winter, pages 187–200, 1995.
- Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Trans. Comput.*, 50(2):131–146, 2001.
- Pattabi Seshadri and Alexis Mericas. Workload characterization of multithreaded Java servers on two PowerPC processors. In *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.



- 20. Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal 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, June 2001.
- T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. In *IBM Systems Journal*, volume 39(1), pages 175–193, 2000.
- 22. Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David P. Grove, and Michael 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, May 2004.
- 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.