JVM SPEC favours 32-bit platforms

Kris Venstermans Koen De Bosschere

ELIS, Universiteit Gent
St.-Pietersnieuwstraat 41
9000 Gent, Belgium
tel: +32 9 264 {33 67, 34 06 }
fax: +32 9 264 35 94
{kvenster,kdb}@elis.UGent.be

Abstract— **The platform-agnostic nature of Java is made possible thanks to the presence of the Java Virtual Machine (JVM) with a well defined set of bytecodes and library routines. This requires the availability of a JVM for many different platforms, 32-bit as well as 64-bit wide. During an effort to port the Jikes Research Virtual Machine (RVM) to the 64-bit PowerPC, we stumbled into a number of problems that prove that the Java specification was designed with a 32-bit platform in mind and that prevent an efficient implementation on a 64-bit platform. All described problems boil down to the fact that all types have a fixed size, while it is in the nature of some types to automatically adapt to the word-size of the current platform. We would like to share those issues with the community, so that future SPEC designers can take into account these flaws. While Java declares to be platform independent, we believe some aspects of the Java SPEC are designed too specific for a 32-bit platform.**

Keywords— **Java, JVM SPEC, 64-bit platform, porting experience**

I. Introduction

The 64-bit world is not new to a variety of back-end server applications, but its growing popularity towards consumer products will make it the major universe to-court in a few years. The main reason that the market for consumer applications is running behind, is that this market is the x86-market and in this segment there is only recently 64-bit hardware available. This is in contrast with the back-end application market, for which special (64-bit) hardware gets developed for many years. Typical applications for which 64-bit machines were build are: data warehousing (large movements of data in memory), scientific computing (calculations on large numbers) and simulation and modeling (large amounts of memory). With the availability of the 64-bit hardware on the x86-market, soon everyone will be confronted with 64-bit applications. One sector that is emphatically waiting for it is that of the large software games.

In order to be able to explore the new opportunities in the 64-bit world, we cooperated on a 64-bit port of the Jikes Research Virtual Machine. It is during this experience that we encountered some limitations of Java that either have a negative influence on the performance, or that force us to write separate code for each platform even when semantic similarity exists.

The rest of this paper is organized as follows. In the next section we'll touch some minor issues, while section III will handle the real limitations and performance bottlenecks. Section IV will present some proposed improvements, which might be helpful for future (Java) SPEC designers to avoid the pitfalls discussed in this paper. Some conclusions in section V will conclude this paper.

II. Some Minor Issues

We first give two examples that are no real problems. They just made us feel a little uncomfortable. The first, about unsigned types, is merely a luxury problem and will not affect the average Java-programmer. It just disallows you to be platform independent when you wish do write some low level applications like a VM. The second has no negative influence on 64-platforms whatsoever, but really favors the 32-bit ones.

A. No unsigned int

Except for char, all types in Java are signed. If you want to use unsigned values, you are forced to wrap them in a signed alternative. This is fine if your range of values is smaller than the positive range of the signed type. Otherwise you must be very careful with the operations you want to perform. This problem arises sometimes for programmers who wants to convert a C-program to Java.

Because Jikes RVM is also written in Java, some tricks were needed to be able to express some low level operations like memory accesses. Addresses, masks, positive offsets are all examples of things that should be expressed

Java types	32 bit platform			64 bit platform		
	Logical size	Cat	Size on stack	Logical size	Cat	Size on stack
boolean	32 bit	I	32 bit	32 bit	I	64 bit
byte	32 bit	I	32 bit	32 bit	I	64 bit
char	32 bit	I	32 bit	32 bit	I	64 bit
short	32 bit	I	32 bit	32 bit	I	64 bit
int	32 bit	I	32 bit	32 bit	I	64 bit
float	32 bit	I	32 bit	32 bit	I	64 bit
reference	32 bit	I	32 bit	64 bit	I	64 bit
returnAddress	32 bit	I	32 bit	64 bit	I	64 bit
long	64 bit	II	64 bit	64 bit	II	128 bit
double	64 bit	II	64 bit	64 bit	II	128 bit

TABLE I
JAVA TYPES WITH THEIR SIZE INFORMATION

program	stack contents (%)			
	int-like	references	longs/doubles	unknown
mtrt	45.89	34.73	1.62	17.74
jess	45.60	34.87	1.60	17.93
compress	46.03	34.37	1.66	17.94
check	46.16	34.32	1.73	17.79

TABLE II
STACK MANIPULATION MEASURED DURING COMPILATION

by unsigned values. Now let us give an example of how careful we must be if we wrap for instance some mask into an int/long respectively on a 32/64-bit platform. Lets say we assign a mask e.g the value 0xFFFF 0000 and that we are interested only in those 16 bits, no matter which platform we're on. If the mask expands to 64 bits on the 64-bit version, the value will become 0xFFFF FFFF FFFF 0000. This is because both values represent the decimal value -65536, the latter sign extended to 64 bits. We have absolutely no choice about sign extension or not, which did not allow us to write platform independent masking operations this way.

B. Dataraces on a 32-bit platform

Because 32-bit machines (generally) do not have 64-bit operations, an atomic update of longs and doubles can be a serious performance issue on such a platform. This is why the SPEC intentionally allows those updates not to be atomic. However, this rule opens the doors for dataraces and is a potential danger for hazards, taken into account that only a minority of the programmers is actually aware of this, and so they might not add the necessary synchronization.

On a 64-bit machine there is no problem whatsoever to atomically update a 64-bit value. In fact it does so always because the 64-bit value is the word size of the machine. The added synchronization around an update of a long/double is obsolete for those machines and might unnecessary slow them down. So finally we could state that what happened here is that a natural thing as an atomic update of a value is made unnatural in favor of the 32-bit platform.

III. PERFORMANCE ISSUES AND LIMITATIONS

A. No natural sized type

The Java SPEC is defined as if all types have a fixed size. However, in practice we experience a whole different world if we shift between 32-bit platforms and 64-bit platforms. Some types (should) automatically adapt to the word size of the machine. What is missing in Java is an integral type of natural size. Probably the best reason to have a natural sized type is for performance. Let's say we define a type that corresponds to the size of the machine word, as long as a minimum size is guaranteed (e.g. 32-bit). Such a type would be ideal for all small ranges used by program-

local index	
0	this
1	int
2	double
3	
4	long
5	
6	float

TABLE III
EXAMPLE OPERAND STACK

mers, such as in for-loops. If we would use another type, then unless the ISA has a set of 32-bit ALU-instructions, most results of ALU-operations will have to be masked to mimic the 32-bit operation. We believe that adding such a type would not break the platform independence of Java as long as we can guarantee that overflow is of no concern.

If you want to write low-level applications that deal with e.g. memory (i.e. a VM, OS), then it would also be helpful to have a natural sized type to represent addresses/references. Now the only solution is to use conditional compilation to map the reference to an int/long respectively on a 32/64-bit platform.

B. 4GiB limitations

Who wants limitations? As long as something is physically possible, it should be allowed as much as possible. Unfortunately, many Java constructs are limited in size by the Java SPEC itself. Those limitations are not visible on 32-bit platforms, because the physical resources have the same scope. On 64-bit platforms, physical borders are stretched much further. The Java SPEC is obvious once more designed with a 32-bit platform in mind.

For example the size of an array is defined as an int, which is bound to a 32-bit value. If we take into account the fact that this is a signed value (which btw is complete nonsense), the largest possible byte-array can never exceed the 2GiB border. This probably is not an acute problem yet, but with the shift towards the 64-bit world this will become an unavoidable issue in time. Maybe a natural sized type could help?

C. Wasted Stack Space

All actual Java types are divided into 2 categories as seen in table I: category I&II computational types. Each category is defined in terms of 1 or 2 stack slots respectively. Now a performance problem arises when we 'd like to make a simple implementation of the Java stack model. A lot of space gets wasted on the 64-bit version. The reason for this is that a reference has become larger, but still has to take one stack slot. So a stack slot will become 64 bits. All other types that do not need this extension, are forced to use the same stack slot definition. So int-like types take one stack slot of 64 bits even though it would suffice to provide 32 bits and longs and doubles will take 2 slots (128 bits) instead of the necessary 64 bits.

In table II , we present some numbers that indicate how the contents of the execution stack is distributed. For each computational type we have a column that shows us which percentage of all stack usage is occupied by this category. We also have an extra column unknown, for the bytecodes that did lose the type information. The information is gathered during compilation of the class files, so it is just for having an idea about the diversification of the use of types by the different bytecodes. On a 64-bit platform, except for references, space is wasted for almost every type. Thus for int-like types, doubles, longs (and some part of the unknown), which count for about 50%, we have half of the space wasted. This would mean a global loose of 25% on average. Those numbers were collected with Jikes RVM 2.2.2 on a linux platform with the BaseBaseSemiSpace configuration. Because Jikes RVM is itself written in Java, these measurements are also influenced and maybe even dominated by the code for the VM itself.

D. Wasted Space for Locals

For the allocation of the locals, we have no other choice[1] than allocating a long and a double twice the space as a computational type of category I. This is so because the Java-to-bytecode compilers automatically increase the local index with 2 units each time they encounter a long or a double. Due to the fact that a reference takes 64 bits on a

[1] speaking for interpreters and JIT compilers that need to be fast. Optimizing compilers can do a better job, but they need to do extra work that isn't necessary on 32-bit platforms

64-bit machine, a long or a double will occupy 128 bits instead of the normal 64. Another consequence is that (even on 32-bit platforms) the use of some specific bytecodes, prohibits the use of other bytecodes. e.g. dload_0 takes local slot 0 and 1, so f.i. dload_1, iload_0 and aload_1 can never be used in the same function.

A little example is in place here. Lets assume we have a method with 4 explicit arguments: an integer, a double, a long and a float. The locals will be numbered as in table III. Index 0 is reserved for the implicit this-reference. The total amount of wasted space on a 64-bit implementation is as follows: 2 entire slots of 64-bits (unused for the long and double) and 2 halve slots for the integer and float (perhaps acceptable to avoid alignment problems, even though alignment problems can be avoided by sorting the parameters). Because the indexes of the long and float are greater or equal to four, all accesses to those variables must go through a wider bytecode of two bytes.

IV. Proposed Improvements

A. New Type Category

If we have an other look at table I, we see two categories of computational types. So far so good: two different categories are probably essential to express some differences between some types. But by which motivations did those two type categories get created? Looking at table I, more specific to the side of the 32-bit platform, we see that all types that do not fit into 32 bits are separated of those that do. However, if we take a look at the other side of table I, it is clear that this is no longer the case on a 64-bit platform. In our opinion it is fine to isolate longs and doubles in a separate category, but to dump all the rest in a single category was less intuitive.

Considering different platforms, it would be a good idea to group together those types that change their size accordingly. So we could create a category III computational type, next to the existing two types of the Java SPEC [LY99]. A reference and a returnAddress then should be a member of this new category III instead of category I computational type. If we introduce a new word sized type to solve the problems of section III-A and III-B, it would also belong in this new category III.

The current Java Specification, has lacked to treat the ”word-sized” types properly. It is because of those types being dropped with the int-like types, that sizes get very unnatural on a 64-bit platform. In the next sections we’ll discuss some steps to take to solve the issues of sections III-C and III-D. We will also come back on the issue covered in this section, namely the need for a third type category.

B. Independent Type Categories

As stated before, the separation of longs and doubles in a separate type category is probably a good thing to do. But worse is the dependencies that still exist between the different type categories. Because the Java SPEC f.i. explicitly says that a type of category I takes one stack slot and a type of category II takes two stack slots, a dependency gets created that states that a category II type always has to take twice the space of a type of category I. We believe it is possible to define all types independent of there actual size (or number of stack slots). If all types are independent of there size, then we can implement the stack model without the overhead of wasting space for almost every type.

B.1 Wasted Space for Locals

Probably the easiest solution to get rid of the double index, used for types of category II is to say don’t use 2 index numbers, just assign one index for each local variable, no matter which type. This would be very convenient for the 64-bit implementation, discussed in section III-D, but would complicate the 32 bit implementation big time. That can not be our intention, so something more general applicable would be appreciated. The cleanest solution to this is in our opinion to number the parameters independent by computational type category.

Let’s check with the bytecodes if this is possible. Well, some examples of bytecodes accessing the local variables are f.i. iload_0, aload_<n>, dstore_2. If we take a look to the complete set of bytecodes that can access the locals, we notice that they all include the type of the variable they want to access. This means that they also implicitly know the type category. So there are no further complications that stops us from using the new way of indexing.

With this new way of indexing, the semantics of a dload_3 would then be ’load the 3th local variable of computational type category II’. This way it is also no longer the case that the use of one bytecode obstructs the use of another. We can use the dload_0 as well as the dload_1, aload_0, iload_1, ..., what leads to possible better usage of shorter bytecodes which will almost certain lead to shorter class files.

We could use the new indexing technique, for starters, only on the existing 2 type categories. This way we would recycle (on 64-bit machines of course) the wasted second stack slot for locals of type double and type long. If we agree on introducing a third type category, proposed in section IV-A, we could go further and store the locals of the remaining type category I on 32-bit slots. The worst case will then be the remaining of one 32-bit slot of waste for the purpose of alignment.

B.2 Wasted Stack Space

The previous subsection concerned the local stack space. In this subsection we will focus on the operand stack. We will also start our discussion in assumption we have two type categories. How can we make both categories size independent ? The condition that has to be met to make this possible, is that each bytecode that accesses the operand stack has to be defined in terms of manipulating (a subset of) only one computational type category.

If we take a look at the bytecodes, we notice most of them are argument specific (e.g. aaload, fstore, dmul) so the condition is met here because they explicitly know the type of the argument, which is more strict than only knowing the computational type category. Unfortunately, it is sadly to notice that the Java SPEC is once again not consistent. This approach is not pursued for all the bytecodes. E.g. the dup bytecode family implicitly takes into account the number of stack slots. A nicer approach would have been to make separated variants for each computational type category. So it really is the bad design of the dup bytecode family which implicitly force longs and doubles to take 2 stack slots.

So in our approach we would need to introduce some extra bytecodes to make separate variants for each type category. This is not a great problem, because there are only about 200 out of 256 possibilities used. We will now consider each bytecode of the dup family and redefine and extend them, as we think should make a better specification. Following the Java SPEC, dup and dup_x1 are bounded to category I computational types [LY99]. So we do not need to change the semantics of those bytecodes. Bytecodes dup_x2, dup2 and dup2_x1 have 2 forms and will be bound to the form with all category I computational types. For the abandoned form, we will introduce 3 new bytecodes: dup_xc2, dupc2 and dupc2_x1 resp., where the c2 indicates one type of category II in contrast to a single number 2 which indicates 2 times a type of category I. Finally the bytecode dup2_x2 has four forms. We bound it to the form with all computational types of category I and introduce three new bytecodes to cover the three remaining forms: dupc2_x2, dup2_xc2, dupc2_xc2.

So with 6 extra bytecodes, we can make the type categories I and II independent of each other's size as far as the operand stack is concerned. With this little adjustment, it is no longer compulsory to give e.g. a double twice the stack space as e.g. a reference.

What in case of the third computational type category? Well in practice it will probably not be the case that, on the 64-bit machine, every type takes its minimal space. Due to alignment problems is it acceptable on a 64-bit platform to allocate 64-bit stack slots for every type. So by splitting the first computational type category, we will not be able to allocate the new category I types on 32-bit slots. So we could state that for practical reasons, we have no need for making the first and third type category independent of their size. But theoretically we think it would have been a cleaner design strategy to do so. So let us once again go to the bytecode level and see what needs to be done to do create the independence between the first and third type category. Of course the dup bytecode family now needs to be extended further to cover all three type categories. Besides this, only three other bytecodes needs to be extended, namely ldc, ldc_w and swap. Those three bytecodes do need to get an equivalent for the new type category.

V. Conclusions

In this paper we wanted to show you some flaws in the Java Specification concerning platform independence. We would like to bring under attention the missing of a natural sized integral type and perhaps also an unsigned integer. Second we would like to emphasize the inconsistent design of the bytecode set, which leads to difficulties regarding efficient implementations of the stack model on different platforms. With the proposed improvements to the Java SPEC, each implementation is now free to decide how much space each category takes, because all bytecodes will now at least know the category of the types it is manipulating. So an 32-bit implementation could e.g. choose to allocate 32-bit for category I and III and 64 bit for category II. A 64-bit implementation could for example allocate 64 bit for every category, without the need of sophisticated compiler analysis (which is the case now if you want to achieve the same performance gain).

References

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman, 2nd edition, 1999.