Java SPEC favors 32-bit platforms

Kris Venstermans, Koen De Bosschere

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. We believe however that some aspects of the Java SPEC are designed too specific for a 32-bit platform.**

Keywords— **Java, JVM SPEC, flaws, 64-bit platform**

I. Introduction

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. This imposes some performance limitations on e.g. the execution stack and limits the scalability of certain Java constructs.

II. 4GiB border

Who wants limitations? As long as something is physically possible, it shouldn't be unnecessary limited by the Java SPEC itself. 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. Those limitations are not visible on 32-bit platforms, because the physical resources have the same scope. On 64-bit platforms, however physical borders are stretched much further, and with the shift towards the 64-bit world this limitation will become an unavoidable issue in time.

TABLE I
Java types with their size information

Java types	Cat.	32 bit platform		64 bit platform	
		Logical size	Size on stack	Logical size	Size on stack
int-like	I	32 bit	32 bit	32 bit	64 bit
float	I	32 bit	32 bit	32 bit	64 bit
reference	I	32 bit	32 bit	64 bit	64 bit
returnAddr	I	32 bit	32 bit	64 bit	64 bit
long	II	64 bit	64 bit	64 bit	128 bit
double	II	64 bit	64 bit	64 bit	128 bit

K. Venstermans and K. De Bosschere are with the Department of Electronics and Information Systems (ELIS), Ghent University (UGent), Gent, Belgium. E-mail: kris.venstermans@elis.UGent.be, kdb@elis.UGent.be .

K. Venstermans is supported by a grant from the Ghent University Special Research Fund

III. 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. On a 64-bit platform, except for references, space is wasted for almost every type. Thus for int-like types, doubles and longs, we have half of the space wasted.

A. New Type Category

By which motivations did those two type categories get created? Looking again 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.

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. Considering different platforms, it would be a good idea to group together those types that change their size accordingly. So we should create a category III computational type, next to the existing two types of the Java SPEC [1]. A reference and a returnAddress then should be a member of this new category III instead of category I computational type.

B. Independent Type Categories

So getting back to the current Java SPEC, the separation of longs and doubles in a separate type category is probably a good thing to do. But worse are the dependencies that still exist between the different type categories. Because the Java SPEC e.g. 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 their size, then we can implement the stack model without the overhead of wasting space for almost every type.

B.1 Wasted Space for Locals

For the allocation of the locals, 2 local slots are allocated for a long and a double. 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.

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 , but would complicate the 32 bit implementation considerably. 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 e.g. 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 stop 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". Another implication of the double index 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 e.g. dload_1, iload_0 and aload_1 can never be used in the same function. By our new way of indexing, we can use them together, what should lead to a better usage of shorter bytecodes which will almost certain lead to shorter class files.

B.2 Wasted Execution Stack Space

The previous subsection concerned the local stack space. In this subsection we will focus on the operand stack. We will start our discussion in assumption we have two type categories. On the operand stack we could save space if both categories are size independent. The condition that has to be met to make the size independence 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 sad 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. 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. Dup and dup_x1 are bounded to category I computational types [1], so we do not need to change their semantics. 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. Besides the dup bytecode family that now needs to be extended further to cover all three type categories, 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.

IV. Conclusions

In this paper we have shown you some flaws in the Java Specification concerning platform independence. 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, an implementation has more freedom in choosing sizes for every computational type category. 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

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