Migrating Bee Smalltalk to a Different Instruction Set Architecture

An Experience Report on Porting a Dynamic Metacircular Runtime from x86 to AMD64

Javier Pimás
Palantir Solutions SRL
Buenos Aires, Argentina
jpimas@palantirsolutions.com

Abstract

We report our experience in porting Bee Smalltalk Dynamic Metacircular Runtime (DMR) from the 32-bit Intel-x86 instruction set architecture (ISA) to AMD64 as a first step to move forward towards a multi-platform Bee Smalltalk.

This port required subtle changes in most areas present in typical Virtual Machines (VMs): low-level object shape, JIT-compiler, garbage collector, primitives and the foreign-function interface. We present a comprehensive analysis of the migration difficulties, and the key implementation and design decisions taken during our work in the context of Bee, which is implemented in terms of a Smalltalk DMR, in contrast to VMs written in languages like C/C++. Additionally, we depict the image-level mechanisms we deviced in order to support the transition between 32 and 64-bit images, which can also be applied to traditional-VM based Smalltalks.

Keywords runtime, virtual machine, processor architecture, porting

1 Introduction

Bee [8] is an implementation of Smalltalk that runs without what is usually known as a VM. Instead, Bee is supported by a dynamic metacircular runtime library written in Smalltalk, which provides all the mechanisms that would usually be implemented by the VM: JIT-compiler, memory management, primitives and its foreign-function interface. Bee DMR is bootstrapped from a derivative of Digitalk Smalltalk running on top of a host VM. The first iteration of Bee was able to run on 32-bit Intel x86 Windows systems, and this paper presents our experience in porting it to 64-bit AMD64[1] Windows.

To Bee's development team, there were two main reasons for migrating from 32 to 64 bits: compatibility with 64-bit applications and the possibility of using more memory.

In this experience report we

- Analyze which parts of Bee Smalltalk, as an archetype of Dynamic Metacircular Runtimes (DMRs), are dependent or independent on the processor architecture.
- Provide the details that let Bee Smalltalk components which depend on the processor architecture vary accordingly.
- Present an iterative migration approach that was successfully applied to cross compile it to another processor architecture, AMD64.

The process of porting from 32 to 64 bits involved solving multiple issues. We give a brief summary of them now:

Object format. The layout of objects in memory in 64 bits does not need to be the same than the one in 32 bits. Our decisions regarding object format are explained in sections 4.1 and 4.5.

Bootstrapping. To create a 64-bit system, it is necessary to generate a 64-bit image, packaged in a 64-bit executable. We detail these issues in sections 4.2 and 4.3.

Native-code generation. Besides objects, a Smalltalk image contains code. In particular, Bee kernel image contains native code. In order to create that native code, it was required to create an AMD64 assembler, and to plug it to the Smalltalk-to-native compilers. Those issues are addressed in sections 4.6 and 4.7.

Integer representation. The size of the word in the system affects how big the small integers can be, and required adapting code, as shown in section 4.8.

Foreign-function interface. The differences in 64-bit Windows calling-convention design affected the way Smalltalk code communicates with external code. It required changing how external functions are called from Smalltalk and how Bee Smalltalk handle external callbacks from C code. This is detailed in sections 4.10 and 4.11.

Chasing platform-dependent code. A part of the migration work is chasing the remaining places where the code is dependent on the processor architecture. For example, this includes code that in traditional VMs is implemented as primitives and in the garbage collector. We describe this in sections 4.9, 4.12 and 5.

[1] also known as x86-64 or, more succinctly, x64

IWST'18, September 10–14, 2018, Cagliary, Italy
2018.

2 Bee Dynamic Metacircular Runtime Overview

Bee DMR is a Smalltalk implementation that runs without a traditional VM. It implements a just-in-time and ahead-of-time compiler, not supported by a host VM. It is the Smalltalk runtime itself that supports the Smalltalk environment, in a similar way than the Smalltalk metamodel is used to describe itself. This self-hosted runtime implementation approach is not new, as it has been previously explored in proyects like Klein [10], a Self implementation, and also in Jalapeño/Jikes [1], and Maxine [12] java runtimes, just to cite some examples.

The main difference between Bee runtime implementation and a traditional Smalltalk VM is that Bee has no primitives. Instead of using primitives, in Bee the programmers can directly or indirectly alter the semantics of Smalltalk code. In particular, the semantics of message sends can be modified arbitrarily. This allows, for example, to implement micro-operations known as *underprimitives*, which can be used to replace primitives. Additionally, the programmer can directly communicate with the native-code compiler to alter the shape of the emitted code. For example, it is possible to select which methods to optimize, which ones to inline or which selectors to change from dynamic dispatch to static invoke. The code that is needed for replacing primitives is, by most parts, not different to the one used to write plain Smalltalk-application code.

All of Bee runtime, including its native code compiler and the memory manager are written in Smalltalk. Consequently, the migration of Bee to AMD64 platform only required working with either Smalltalk code or either assembly code.

3 Differences between x86 and x86-64 platforms

AMD64 [5] is the 64-bit version of the 32-bit x86 architecture. It supports wider memory addresses (up to 64-bits in theory) and 64-bit registers and operations. Typical x86 registers have expanded to add 64-bit versions of them. Additionally, 8 new 64 bit registers are available (R8 to R15), as shown in figure 1. x86-64 instruction set is mostly backwards compatible with x86, in the sense that most instructions present in x86 are also present in x86-64, and are encoded in a similar way in both architectures. In many cases, x64 instructions analogous to their x86 counterparts are encoded exactly the same as in x86; in other cases, the x64 version requires adding prefixes. This is exemplified in figure 2.

3.1 Calling Convention and Application Binary Interface Changes in Windows

In Windows-x86, there are two main calling conventions supported: *stdcall* and *cdecl* [7, 11]. They are similar, with EAX, ECX and EDX as callee-saved registers and arguments pushed into the stack from right to left. The most notable difference is that in cdecl the caller is responsible for popping the arguments out of the stack, while in stdcall this responsibility is assumed by the callee.

x86	AMD64		
32-bit	**64-bit**	**32-bit**	**64-bit**
eax	rax	*r8d*	*r8*
ebx	rbx	*r9d*	*r9*
...		...	
esp	rsp	*r14d*	*r14*
ebp	rbp	*r15d*	*r15*

Figure 1. To the left, the original x86 general purpose register set. To the right, the registers added in AMD64. For each x86 32-bit register, a 64-bit counterpart has been added; 8 new 64- and 32-bit registers were also added.

instruction	**encoding x86**	**encoding x64**
push ebp	55	-
push rbp	-	55
mov eax, ecx	89 C8	89 C8
mov rax, rcx	-	48 89 C8

Figure 2. Instruction encoding is kept similar. For some instructions like push, the same encoding is decoded differently in x86 and AMD64, to adapt to the new word size. For others like mov, specifying a 64-bit register requires adding a prefix.

On the other hand, in Windows-x64 only cdecl convention is supported. The 64-bit version of cdecl, however, has a few differences compared to the 32-bit one: caller saved registers are RAX, RCX, RDX, R8, R9, R10, R11; the first 4 arguments are passed in registers RCX, RDX, R8 and R9; there is a *shadow space* preallocated in the stack before the call and the stack is 16-byte aligned considering the arguments at the instant before the call.[2] Calling conventions directly affect the foreign-function interface implementation, because they influence the native code needed to perform external calls, as explained later in section 4.10.

In Windows, except for pointer data types, C types mainly maintain their size. This is particularly true for int and long data types, which are kept 4-bytes wide. However, the change in pointer size affects the layout of structure fields in memory, as fields which are placed after a pointer in a C structure will see their offset increased. Data type size affects C structures and, as a consequence, also affects the foreign-function interface implementation.

Executable files and dynamically-linked libraries for Windows are stored in a format known as Portable Executable

[2] Unlike in System-V/x64 where it is aligned before calls without considering the arguments

(PE). Files in this format contain not only the data and executable code of the program or library, but also a set of tables that describe what is stored in the file, such as exported function offsets, imported functions and relocation information. When 64-bit Windows was created, the PE format was expanded to support 64-bits executables. The new format, PE32+, is mostly equal to PE32, except for a few tables which store pointers, that were widened to 64 bits.

4 Bee Execution Model and Migration to AMD64

The first step required to start the transition from x86 to AMD64 was to discover which parts of Bee were dependent on the processor architecture and which were not.

Bee has been designed and implemented in terms of an abstract register machine. This machine design is intended to be agnostic of the underlying processor architecture and word size. It consists of an abstract set of registers and operations that are converted to native code and data according to a concrete target architecture. Furthermore, Smalltalk images usually do their best to be independent of the running platform, improving portability.

However, in any Smalltalk system there are places where the processor architecture surfaces, as in the representation of numbers, in the foreign-function interface, and in calls to primitives. This happens in implementations that run on top of a typical VM, and Bee DMR adds to that list the set of components that implement the low-level interface to the platform. As an example, for x86, a set of compiler objects were implemented that allowed for native x86 code generation at different stages (inline assembly, baseline JIT, optimizing JIT). These objects create a layer of separation between code specific and independent of the architecture. Yet, during migration we noticed that in practice there were other spots where the x86 architecture details leaked into the runtime code. As Bee is a dynamic metacircular runtime, there is not a clean separation of VM code and guest-language code, which in typical Smalltalks is mostly target-agnostic.

4.1 Migration Approach for the AMD64 port

To facilitate the implementation of the AMD64 port, we chose to take an incremental development approach. We decided to make the minimal amount of modifications possible in order to have the AMD64 version fully functional in the shortest possible amount of time. These modifications were basically two:

- Widen the slots of objects from 32 to 64 bits.
- Implement AMD64 code generators.

In contrast, there were a set of design decisions *we chose not to change* at the same time, among which were the object header format and the garbage collection algorithm [9]. We considered that while widening to a 64-bit word size *lets* to implement more efficient algorithms in areas like GC, mixing these changes with the ones needed for the 64-bit port would lead to unnecessary instability of the system. We believe that those changes can be done in ulterior stages. Limitations of our approach are described in section 5.2.

The two modifications we decided to carry out had diverse implications on the different components of Bee runtime. In the following paragraphs we provide a detailed description of them.

4.2 Bootstrapping

In order to establish the 64-bits system, we do not convert objects to 64-bits online, but instead we setup the bootstrap mechanism and Smalltalk library writers to cross-compile a 64-bit system from the 32-bit one. This generates a new 64-bit executable that when executed will already live in the AMD64 world. It is not possible in Bee to use 32 and 64-bit objects at the same time. The 64-bit objects are serialized by the library writers at the cross-compilation step.

During this writing process a few objects need to be supervised:

- A set of Smalltalk globals are adapted according to the size of the target architecture. For example the global WordSize is set to 4 or 8 accordingly.
- SmallIntegers are migrated to the target word size.
- Methods for accessing external pointers are chosen and installed depending on the target word size.
- Classes representing external structures are updated to use the correct field offsets, as explained in 4.10.

At the same time, a few initialization steps were added, so that constant objects are set to the appropriate values at launch time. For example, things like null ExternalAddress or ExternalHandle have to be created with the according amounts of bytes.

4.3 OS Executable File Format

In typical Smalltalk VMs, which are written in or translated to C/C++, the executable code of the VM is stored into PE files by the C/C++ compiler, which understands PE and can output executables in that format. In Bee, the executable runtime image is bootstrapped from a set of objects that represent code and data. Those objects are packed into a PE file by a mechanism that models the PE format in Smalltalk. For this reason, the migration of Bee DMR to x64 platform started by the implementation of the PE32+ format.

4.4 Change in Word Size

In Bee, as in any Smalltalk, the big majority of the classes are independent on the size of the word. Only classes that require implementing lower-level components need adjusting to the system word size. The most notable ones in Bee were Process, Thread, Memory, StackFrame, ExternalHandle, ExternalAddress, FFIMethod, SmallInteger and LargeInteger.

In Bee, there are only two types of objects in the heap: byte objects and pointer objects. Pointer object slots are accessed indirectly via instance variable reads and writes (emitted by the JIT), or directly through #_basicAt: and #_basicAt:put: underprimitives, explained in section 4.9. In both cases, *the offsets calculated for object or stack slots are never written explicitly*, but computed by the JIT compilers or the implementation of #_basicAt:(put:). This meant having single points of modification for adapting to the desired word size. The implementation of offset calculation for instance variables was adapted by only having to touch the JIT and a couple of low-level methods, and this was enough to adapt slot accessing in pointer objects. Byte objects, on the other hand are mostly independent of the word size, except on two cases: when handling large integers, and for objects that represented addresses, like ExternalAddress. However, in those latter cases, the changes needed were small: switching from using the constant 4 (for the word size) to using the global value WordSize, or in other cases, using a constant associated to the word size (i.e. the maximum SmallInteger). Those constants are stored in pool dictionaries, which can either be changed during bootstrapping or either be initialized at start-up time.

Finally, we implemented a set of objects that represent the different application binary interfaces (ABI) in use: X86ABI and X64ABI. These objects provide the knowledge needed to adapt code generators to the underlying architecture. For example, they provide the mapping from abstract registers to concrete ones. There are 8 classes that use those objects, all used by the different native-code compilers: the baseline JIT, the register allocator, the assembly-code emitter, and other stages of code optimization.

4.5 Bee Object Memory Format

In Bee's memory, objects were stored in a format designed for 32-bit arquitectures [8]. In that format, objects contained an 8-byte or 16-byte header that specified the object size, its type and some other properties like whether they contained pointers or bytes. Our desire to allow for using more memory impacted directly in the format of objects in memory: the most straightforward way of allowing this is to widen object pointers to 64 bits, which in our case meant to make slots of objects in memory double in size.

4.6 Assembly Encoder

Bee contains two main native-code compilers: a baseline JIT and an optimizing compiler. Both of them use the same assembler as a back-end which has been designed in terms of the abstract register machine previously mentioned. In this machine, there exist R register for passing a receiver or returning a value, A register used mostly for storing an argument, T register for a temporary, S for storing self, E for the current closure environment and finally SP and FP for the stack top and frame pointers respectively. This results in 7 registers which is almost the same amount of general purpose x86 registers.

```
X86ABI>>#regR
  ^eax

X64ABI>>#regR
  ^rax

BaseAssembler>>#and: op1 with: op2
  self encode: 'and' with: op1 with: op2

BytecodeAssembler>>#andRwithA
  self encode: 'and' with: abi regR with: abi regA

BytecodeAssembler>>#compareRwithSindex: index
  pointer
    reset;
    length: abi addressLength;
    base: abi regS;
    displacement: index − 1 * abi wordSize.
  self encode: 'cmp' with: abi regR with: pointer
```

Figure 3. X86ABI and X64ABI answer a different R register. The assembler delegates slot indexing to the abi object as much as possible.

In x64, the number or registers has been doubled, but Bee abstract machine has been kept without major modifications. The concrete registers used in x86 are replaced with their expanded 8-byte counterparts. The assembler instruction encoding interface works at two levels: on one hand, it provides methods to encode instructions passing concrete registers; on the other hand, it provides an API to pass abstract registers, which are mapped one-to-one to concrete ones according to the target platform. For both of those two levels, the assembler API does not directly expose x86 or x86-64 instructions. Instead, operations provided to the client of the assembler interface are more abstract. Examples of these operations are things like pushing and popping values into and out of the stack, loading and storing values from and to memory, or performing arithmetic and logical operations in registers.

Figure 3 shows the implementation of typical methods of the assembler interface, and how the a assembler uses the ABI objects described in 4.4 to abstract away the differences between x86 and x64. Those objects provide the set of available registers in the platform, a mapping from abstract registers to concrete ones and the target word size. They also allow the assembler to transform from pointer indexing operations to slot offsets according to the word size.

x86 instruction encoding is a complex process. The assembler delegates this task to another object: the InstructionEncoder. This encoder is in charge of writing the instruction

prefixes, opcode and operands into a machine code stream. Notably, the same encoder can write both x86 and AMD64 instructions, because their encoding is similar.

The new assembler interface. Implementing the new x86-64 assembly encoder was the most time consuming task of all the migration project. The main reason was that we did not do a direct port from the x86 assembler to an x86-64 one. The original x86 assembly encoder was targeted to be used mostly by the JIT compiler. This meant that its API only provided for instructions used by the JIT, which were directly translated to the bytes that encoded those instructions. For that reason, native-code stubs needed for callback and dispatch optimization mechanisms could not be written using our original x86 assembler, and were hard-coded as byte arrays. Those arrays were created by writing and assembling them with external tools. The new assembler was designed to be more generic so that we could change to dynamically generated stubs, written in terms of Bee abstract machine, working in both architectures. The greatest challenge on this area was implementing the logic behind instruction encoding in x86-64, which is very complex. Additionally, the new assembler is capable of encoding instructions for both x86 and x86-64 modes.

Instructions with 64-bit immediates. x86-64 operand encoding presents a subtle but important limitation compared to x86. While in x86 it is possible to encode immediate values of the word size (32 bits), in x64 immediate operands are limited to only 4 bytes. The only exception to this limitation is in movabs instruction, which allows encoding 8-byte immediate values into 64 bit registers. This limitation makes it harder to manipulate pointer and small integer values in JIT-compiled code. In x64 we added an abstract V register, which is used to overcome this limitation. When needed, operations with 8-byte immediates (i.e. pushing a pointer in 64 bits) are done in two steps: first the immediate is moved to V register using movabs, then V is pushed. V register is mapped to R11 in x64. This approach is not the only possible one. Another solution is to store pointers separate from native code, and to only let native code indirectly manipulate them, through some base register. For example, in x64, it is possible to use RIP-relative addressing, so a pointer table could be stored after each method's native code. In native code, pointers could then be accessed via instructions like mov rax, [rip+k] or push [rip+k], where k is an offset from the instruction to an entry in the table. We did not implement that solution for two reasons: such a change would be against our minimal modifications approach (section 4.1), and also because RIP-relative addressing is not available in 32-bit x86, so we would have needed to use different pointer encoding techniques for x86 and x86-64. Yet another possibility would be to split pointer loading by combining smaller 32-bit bit-shift and bit-or operations. We discarded that technique as we considered it too complex. For example, the garbage collector would need logic to reconstruct each object pointer in JIT-compiled code, as the original pointers would be split in many instructions.

4.7 Native-Code Compilers

The baseline JIT communicates with the assembler in terms of abstract instructions and registers. This makes its transition to 64 bits mostly transparent. Even the registers used are abstract, and the assembler translates them to concrete ones.

For the optimizing compiler, on the other hand, there is almost no mapping from abstract to concrete registers. This compiler directly talks to the target ABI objects described in section 4.4, to obtain the set of available registers in the architecture. It also asks for the concrete registers assigned to the receiver and the return value. With all that information in hand, the register allocator can assign registers to the values of its intermediate representation and delegate encoding to the assembler. Intermediate operations, on the other hand, are target agnostic. As a last step, the machine-code emitter talks to the assembly encoder who converts them to x86 or x64 instructions accordingly.

Figure 4 shows various snippets of the native-code compilers. The assembler interface which abstracts away concrete register names, used by the baseline JIT, is shown in the first example. The assembler, according to its own configuration, is in charge of converting abstract registers to concrete ones. To the client, the assembler interface does not expose specific instructions of the x86 or x64 architectures. Actually a single operation, from the client point of view, could be implemented by the assembler as a list of concrete machine instructions. In the last example we see the code emitter for the optimizing compiler. In this case, the other level of the assembler is used. The intermediate operations are assigned two concrete registers, and finally the assembler is told to emit machine code to apply a bitwise or on them.

4.8 Integer Representation

Bee integers are divided in SmallIntegers and LargeIntegers, as usual in many Smalltalks. Both types of integers are stored in a two's complement representation. The code of those classes is dependent on the word size but mostly independent of the processor architecture (specially because of the use of the abstract assembler). The methods had to be reviewed so that they would adapt to the word size. Example of this were SmallInteger»sizeInBytes, SmallInteger»bitShift: or LargeInteger»reduce.

4.9 Primitives and Underprimitives

Unlike traditional Smalltalk VMs, Bee DMR has no primitives. The code for what is usually represented as a primitive is instead coded in Smalltalk, using *underprimitives*, which can be seen as very specific fragments of primitive operations

```
LoadArgumentBytecodeNativizer>>assemble
  assembler loadRwithFPindex: self argumentIndex

JumpBytecodeNativizer>>assemble
  | target |
  target := methodNativizer labelAt: self target.
  assembler jumpTo: target

LookupLinker>>emitSend: aSelector using: anAssembler
  | send |
  send := SendSite sending: aSelector with: lookup.
  anAssembler
    load: anAssembler regA withPointer: send oop;
    holdJustEmmitedReferenceTo: send;
    callIndirectReg: anAssembler regA

BareMetalCodeEmitter>>assembleBitOr: instruction
  | left right |
  left := allocation at: instruction left.
  right := allocation at: instruction right.
  assembler or: left with: right
```

Figure 4. Typical snippets of the different code generators communicating with the assembler.

like accessing raw pointers in memory, or doing low-level arithmetic and logical operations on registers. The code of what used to be primitives is written in Smalltalk. The only trick is that at specific points, for special messages sends, the JIT compiler inserts special inline assembly code instead of doing a lookup dispatch. The methods that replace primivitives end-up being mostly independent of the architecture, and the low-level underprimitive details were the only things that required adapting. In this port, as the assembler already provided the abstract interface, the only requirement was to take into account that the word size could be 4 or 8 bytes.

In the case of the object headers, as they remained unchanged, few modifications were needed. The access to the bits present in object headers is reified in the class ObjectHeader. This serves as a single point of modification if any change to the header format is done. For byte accesses (i.e. object flag accesses) there was nothing to do. For slot accesses, used for behavior and extended object size, care had to be taken to always use 32-bit wide reads and writes. Otherwise, when targeting 64-bits, the assembler would incorrectly use a slot size of 64-bits to access these 32-bit fields.

Figure 5 shows an example of a part of at: primitive implementation. basicObjectAt: is the part used for accessing slot objects. It does not contain any platform dependent code. basicObjectIndexOf: receives a slot index, an returns other one (depending on the object type it could be different of the received one). Finally, _basicAt: underprimitive is inlined by the JIT to actually access that slot in loadRwithRatA. The

```
Object>>#basicObjectAt: grossIndex
  | index |
  index := self basicObjectIndexOf: grossIndex.
  ^self _basicAt: index

InlineMessageLinker>>#assembleBasicAt
  | nonInteger |
  nonInteger := assembler labeledIntegerNativizationOfA.
  assembler
    loadRwithRatA;
    @ nonInteger
```

Figure 5. A part of the implementation of at: primitive.

slot offset is calculated at execution time, by shifting the index 2 or 3 bits according to the word size.

4.10 Foreign-Function Interface

Foreign-function interface of VMs usually comprises calling external C functions, accessing external C structures being called by external code through callbacks. In Bee DMR, all of those things are implemented in Smalltalk.

The calling convention for Windows x64 is cdecl, which works in a very similar way to its 32-bit x86 counterpart. The first 4 arguments (from left to right) are not passed in the stack but through registers. However, the convention demands a shadow stack space of the same size than those 4 registers. This fact helped us to reuse the 32-bit cdecl code, which passes all arguments through the stack. For the 64-bit version, we just add a final step before the call to the external function, to offload the contents of the topmost 4 stack slots to registers.

Support for accessing external C structures in the different architectures did not present big obstacles but required to solve a subtle discrepancy: for a same C structure, the memory representation can vary depending on the processor architecture and the platform. This means that the size and offset of the fields in the structures can vary depending on the platform.

Bee resolves this problem mostly automatically. First, all C structures are represented with classes that implement the class-side method def. This method returns for each class the corresponding C structure definition, as a string. During development time, a parser reads those definitions and generates accessor methods for each field with the correct size and offset. The offsets are specified through pool variables. The parser generates two analogous pool dictionaries. The keys in those dictionaries are the names of the fields in the definition method, and the values are their corresponding offsets, for 32 bits in one dictionary and for 64 bits in the other. Field offsets are never specified using constants but using their corresponding automatically generated pool variable.

```
typedef struct tagCOPYDATASTRUCT {
                          // off-32 | off-64  |  size
  ULONG_PTR dwData;  //       0        0       4/8
  DWORD     cbData;  //       4        8       4
  PVOID     lpData;  //       8       16       8
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

```
COPYDATASTRUCT>>cwData:
  self longAtOffset: cwData

COPYDATASTRUCT>>dwData: anInteger
  self pointerAtOffset: dwData put: anInteger
```

Figure 6. A C structure of Windows. The size of dwData is 4 or 8 bytes. The offset of cbData and lpData is different in 32 and 64 bits. lpData offset is affected by padding in 64-bits. The total size of the struct is 12 bytes in 32 bits and 24 bytes in 64 bits. A parser calculates all this automatically and generates accessor methods.

When bootstrapping the system, the library generator iterates through all classes that correlate to external C structures and accordingly tweaks them to use the pool dictionaries that correspond to the target architecture. Figure 6 shows an example of a Windows C structure, with the field sizes and offsets. The snippet also shows two of the automatically generated methods to exhibit how the generated code adapts to the varying sizes and offsets.

From the point of view of the clients of the foreign-function interface, little care to processor architecture is needed when calling external functions. The generated structure bindings contain functionality to allocate them in the external C heap. Calculation of the correct structure size is abstracted from the client. A manual code review of the clients of the foreign-function interface was still required, to detect places where external pointers were incorrectly assumed to be 4 bytes.

4.11 Callbacks

For C callbacks, adding support for x64 required only little changes. When a callback is received by Bee, a native-code stub saves the processor registers according to the calling convention and sends a Smalltalk message to the object that owns that stub. As the last step of the callback handling, the processor registers are restored and a return is issued. The original callback stub was written in assembly, and the encoding of the assembly instructions was stored in a byte array. We changed this to use the new abstract assembler interface. The code is generated using abstract register names, which makes most of the code independent of the processor architecture. Between x86 and x86-64, there are two main differences though: in x86-64 some arguments are passed in registers; as x86 defaults to stdcall calling convention while x86-64 defaults to cdecl, stack clean-up has to be done a little bit differently. The difference in argument passing between x86 and x86-64 is solved in a way analogous to what is done for C function calling. At the prologue of the callback stub, the registers RCX, RDX, R8 and R9 are copied back to the stack in the shadow space. This allows the callback handling code that follows the stub to be able to read all arguments from the stack independently of whether the platform is 32 or 64 bits. On callback exit, the only difference is that for 32 bits the callback clean-up code has to pop the arguments from the stack, while in x64 the arguments are popped by the caller.

4.12 Garbage Collection

Adjustments needed for Bee's garbage collector [9] have been minimal. Of the 83 methods that compose the garbage collection algorithm, only 5 needed any change. There were two types of modifications:

Behavior Slot. Code that dealt with the behavior slot in object headers had to be tuned, because generic reads and writes of slots (which were used originally in 32-bits code) would become 64-bit memory accesses in x64. Instead, as the behavior slot in object headers is only 32-bit wide, a special 32-bit access has to be done both in x86 and x64.

Forwarding Index. In different phases of Bee GC copying algorithm, object addresses are converted to forwarding indexes. This is done by subtracting the base address of a GC space to the object address, and then shifting the result 2 or 3 bits, depending if 32 or 64 bits. For that, the WordSizeShift global was used.

5 Discussion

Bee DMR is fully written in Smalltalk, in contrast to typical Smalltalk VMs written in C/C++, and also different from Squeak's VM which is written in slang but then translated to C [4]. This poses a significant difference between Bee and others: in Bee, the majority of memory accesses are done in terms of type-less object pointers.

One of the most feared challenges predicted before the migration started was the migration from a 32-bit/4-byte architecture to a 64-bit/8-bytes one. The difficulty expected was that detecting all the places where Smalltalk code assumed a word size of 4 bytes would be hard. In practice, this problem did not result as tough as expected. Detection of the methods that required changes was done manually, looking at methods of classes that were suspicious, and also by searching for senders of very low-level messages. In total, around 120 methods in the whole system depend on the contents of the global WordSize. Less than a dozen methods use a similar variable called WordSizeShift, which is used when converting pointers to indexes and vice versa, via shifting

operations. There are 7 classes that use a wordSize instance variable: half of them are used for the assembler/disassembler, the other half for building code libraries.

5.1 Debugging

Debugging the 64-bit version of Bee posed a challenge, specially at the beginning of the process. We used a combination of IDA debugger and disassembler [2] and our own native-code debugger. With both of them, an initial effort had to be done, to make debugging scripts portable to both x86 and AMD64, and to abstract away the differences that objects present in memory.

While the detection of most of the Smalltalk methods that had to be changed occurred during a Smalltalk code review process, a little amount of them were not found initially and were only caught at run-time, when they would end crashing the system.

5.2 Limitations

The migration approach described in 4.1 has limitations:

- The size field in the header of objects is limited to 32 bits, which in practice means that the biggest object can be 32GB.
- Pointers to behavior are 32-bit wide, so behavior objects have to live in the lower 4GB area of memory. This limitation was introduced to ease in migration from 32 to 64 bits, but added a series of drawbacks. As behaviors have to live in the lower memory, special care has to be taken when instantiating behaviors (putting them in a special GC space if needed). Smalltalk libraries also have to be treated specially, loading them below the 4GB limit or migrating their behaviors to the lower area after loading them.
- The GC algorithm has not been modified, and is not optimized for heaps with size in the order of gigabytes or bigger. Currently, the only implemented approach is a scavenging copying collector (which is disabled by default in the old space). Enabling it in such scenarios may introduce undesirable pauses to the system.
- At present, it is only possible to cross compile from x86 to x64, but not the other way around. While going from x64 to x86 should be straightforward, we have not done any attempt to support that functionality yet.

5.3 Lessons

The target-agnostic assembler interface proved useful to build a compilation framework that can freely change between processor architectures. The combination of an assembler API that accepts both abstract registers and concrete ones, with the help of the X86ABI and X64ABI objects, allowed to share the code of all stages of compilation in both architectures.

Smalltalk code is inherently independent of processor architecture, and this extended to the kernel that forms Bee DMR. Components like the assembler required modifications, not because they depended on the architecture, but because they modelled the architecture.

In a Smalltalk image, foreign-function interface is the main platform-dependent piece of code of user applications. If the C bindings are generated automatically, the migration task is reduced to a minimum, only requiring the developer to deal with corner cases. This experience showed the value of adopting a systematic approach for handling code that communicates with external libraries: having a parser for C structure definitions and generates Smalltalk accessors frees the programmer from writing error-prone boilerplate code.

We did not do a comparable port of a traditional VM, so we cannot directly contrast how this work would apply in that situation. However, there are points to be evaluated. We do not see any obstacle to use the same assembler design in a traditional VM. Moreover, at image level, automatic generation of bindings for C structures is directly applicable to any Smalltalk. We still do daily development of Bee on top of a host VM. This posed the advantage of letting us work freely on the native-code compilers and assembler, because they were not in use at the same time they were being modified.

6 Validation

To assess the performance of the 64-bit port we run a series of benchmarks. In particular we used the Are We Fast Yet benchmarks [6], ranging from micro to macrobenchmarks. DeltaBlue and Richards [13] are classic benchmarks evaluating the performance of object-oriented applications. Havlak [3] is an optimization algorithm for a compiler but is representative for many application-level optimization problems, too. And the Json benchmark parses a larger JSON document, which is relevant for the performance of many REST services used in today's web applications or micro services. The rest is a collection of numerical and OO benchmarks stressing particular aspects of the implementation.

The benchmarks were run on a machine with a 2.8Ghz 4-core Core i7 7700HQ with hyperthreading and 16GB of memory. The operating system is a 64-bit Windows 10. Measures were taken collecting 50 iterations for each benchmark.

For the performance comparison, we consider peak performance only and discount start-up, warm-up, and JIT-compilation times.

The benchmarks are run with an initial heap size of 64 MB, to minimize noise introduced by the GC. The results are normalized to the 32-bit version of Bee DMR, to use it as the baseline for the performance comparison. We report averages and confidence intervals with $\alpha = 95\%$.

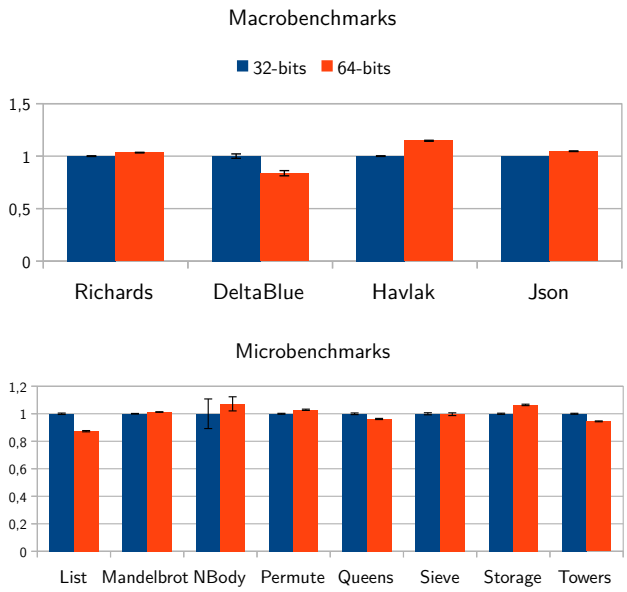

Figure 7. Normalized macro- and micro-benchmarks execution times, relative to 32-bit Bee (lower is better).

The results are shown in Figure 7. We can observe that performance of the 64-bit version is competitive with the 32-bit one. The results are mixed, ranging from taking 12% more time in the worst case (Havlak) for the 64-bit version, to being 16% faster in the best case (DeltaBlue). The algorithms used to implement the 64-bit version of Bee DMR are the same than the ones used in the 32-bits implementation. In 64-bits Bee, pointer objects can be almost double in size than their 32-bits counterparts. However, the AMD64 architecture seems to be optimized to keep up with the increased memory access demands. On the other hand, in 64 bits, small integers can be used to represent bigger numbers than in 32 bits, which could help to improve performance of integer arithmetic operations. Taking all these characteristics into account, the results are not unexpected or surprising.

7 Future Work

The work presented here served the authors as a strong evidence of the usefulness of the migration approach described in section 4.1. It gives us confidence on the potential to use that same approach to expand Bee to other platforms. There remains to be explored how this same design would stand when porting to other not so similar processor architectures like ARM or RISC-V. We still would like to establish what are the obstacles in bootstrapping back from a 64-bit system to a 32-bit one. Finally, now that we have full support for 64-bits environments, we have to discover how to manage the huge amounts of memory that the system can theoretically handle; this will impact mostly in the design of Bee's garbage collector.

Acknowledgments

The author wants to thank Leandro Caniglia, Valeria Murgia, Jan Vrany and the rest of the development team of Palantir Solutions for providing valuable ideas, discussions and reviews, and being in charge of the development and maintenance of Bee runtime libraries. This work was funded by Palantir Solutions.

References

[1] 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. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211

[2] Chris Eagle. 2011. *The IDA pro book.* No Starch Press.

[3] Robert Hundt. 2011. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days* 2011 (2011), 38.

[4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97).* ACM, 318–326. https://doi.org/10.1145/263698.263754

[5] Intel. 2018. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 1: Basic Architecture* 2 (2018).

[6] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16).* ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[7] Microsoft Corporation. 2018. Argument Passing and Naming Conventions. (2018). https://msdn.microsoft.com/en-US/library/984x0h58.aspx [Online; accessed 20-July-2018].

[8] Javier Pimás, Javier Burroni, and Gerardo Richarte. 2014. Design and implementation of Bee Smalltalk runtime. (2014).

[9] Javier Pimás, Javier Burroni, Jean Baptiste Arnaud, and Stefan Marr. 2017. Garbage Collection and Efficiency in Dynamic Metacircular Runtimes. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17).* ACM, 12. https://doi.org/10.1145/3133841.3133845

[10] David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM, 11–20. https://doi.org/10.1145/1094855.1094865

[11] Wikipedia contributors. 2018. X86 calling conventions. (2018). https://en.wikipedia.org/w/index.php?title=X86_calling_conventions&oldid=850925564 [Online; accessed 20-July-2018].

[12] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[13] Mario Wolczko. 1996. Benchmarking Java with Richards and Deltablue. (1996). http://www.wolczko.com/java_benchmarking.html