Optimizations in the Cibyl binary translator for J2ME devices

Simon Kågström, Håkan Grahn, and Lars Lundberg
Department of Systems and Software Engineering, School of Engineering,
Blekinge Institute of Technology
Ronneby, Sweden
{**ska**, hgr, llu}@bth.se

Abstract

The Java J2ME platform is one of the largest software platforms available, and often the only available development platform for mobile phones, which is a problem when porting C or C++ applications. The Cibyl binary translator targets this problem, translating MIPS binaries into Java bytecode to run on J2ME devices. This paper presents the optimization framework used by Cibyl to provide compact and well-performing translated code. Cibyl optimizes expensive multiplications/divisions, floating point support, function co-location to Java methods and provides a peephole optimizer. The paper also evaluates Cibyl performance both in a real-world GPS navigation application where the optimizations increase display update frequency with around 15% and a comparison against native Java and the NestedVM binary translator where we show that Cibyl can provide significant advantages for common code patterns.

1. Introduction

A large majority of the mobile phones sold today come with support for the Java 2 Platform, Micro Edition (J2ME) [17], and the installation base can be measured in billions of units [14]. Mobile phones are also quickly becoming more and more powerful, having processing speed and memory comparable to desktop computers of a few years ago. J2ME is a royalty-free Java development environment for mobile phones, and is often the only available method of extending the software installed on the phone. This poses a severe problem when porting C or C++ applications to J2ME devices, which can often require a complete rewrite in Java of the software, possibly assisted by automated tools [3, 6, 10, 11].

Cibyl is a binary translator which targets this problem. Cibyl translates MIPS binaries compiled with GCC into Java bytecode, and provides means to integrate with native J2ME classes. Cibyl therefore allows C and C++ programs to be ported to run on J2ME phones. When designing Cibyl, our goals have been to produce compact translated code with performance close to native Java code for common cases. Compared to implementing a Java bytecode backend for a C/C++ compiler, the binary translation approach can require less engineering, since Java bytecode (which is type-safe and modeled for the characteristics of high-level Java code) might not be a good match for the compiler structure. The general design of Cibyl has been described in an earlier paper [8], and this paper focuses on optimizations made to reduce the size and improve the performance of the translated binaries.

The optimizations we employ for Cibyl share some similarities with regular compiler optimizations, e.g., use of function inlining and constant propagation, but is also significantly different. Since the GCC compiler has already optimized the high-level C code, the goal of the Cibyl binary translator is to make the translation into Java bytecode as efficient as possible. Because of this, the binary translation optimizations we apply are mostly local in scope, acting on a small set of adjacent instructions or the opposite act on large entities such as entire functions.

The main contributions of this paper are the following. We first describe the set of optimizations we make and how these improve size and performance. We then perform a benchmark on an application ported with Cibyl to illustrate the optimizations in a real-world setting. Finally, we compare Cibyl against native Java and another binary translator, NestedVM [1] in a micro benchmark (an implementation of the A* algorithm) to study performance characteristics in detail. The performance results show that Cibyl is significantly faster than NestedVM and close to native Java performance on the cases we target.

The rest of the paper is structured as follows. In Section 2 we introduce the Cibyl binary translator. The main part of the paper then follows in Section 3 where we describe the optimizations performed by Cibyl and Section 4 where we evaluate our optimizations. Section 5 describes related work and finally we conclude and present future directions in Section 6.

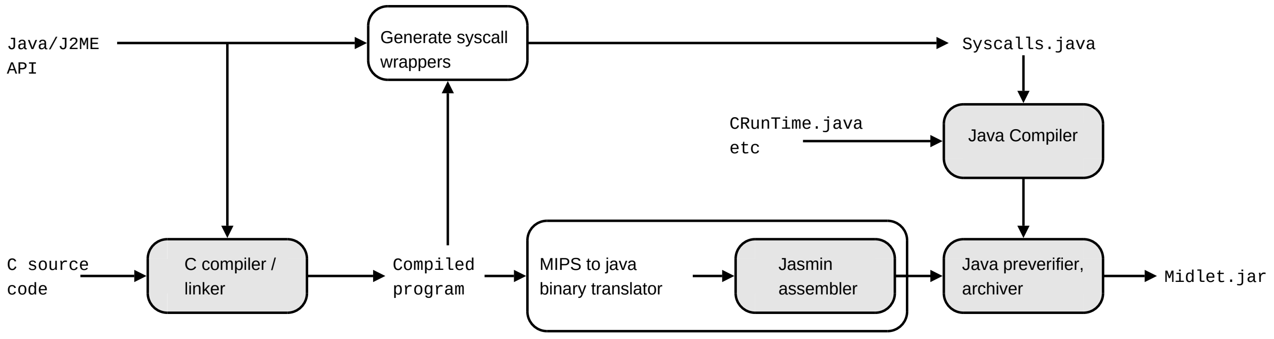


Figure 1. Translation process in Cibyl. Gray boxes show unmodified third-party tools.

2. Cibyl

Cibyl uses the GCC [16] compiler to produce a MIPS I [7] binary, which is thereafter translated into Java bytecode by the Cibyl tools. Figure 1 shows the translation process with Cibyl, where we use a set of tools to translate the MIPS binary. Apart from the binary translator, which outputs Java bytecode assembly to Jasmin [12], we also have a stub code generator to produce stubs for calls into native Java code. When translating, Cibyl uses Java local variables to represent MIPS registers, which contributes to producing efficient and compact code compared to using class member variables or static class variables. The MIPS instruction set is well suited for binary translation, and most arithmetic instructions can be directly translated to a sequence of loading source registers (local variables) on the Java operand stack, performing the operation and storing into the destination register.

We use a Java integer array to represent memory as seen by the MIPS binary. This means that 32-bit memory accesses are performed efficiently by simply indexing the memory array with the address divided by four, but also that 8- and 16-bit accesses need an extra step of masking and shifting the value in memory to get the correct result. Since a common pattern is to use the same base register repeatedly with different offsets, we pre-calculate the array index and use special memory access registers for these cases. To reduce space, we also perform the more expensive 8- and 16-bit accesses through functions in the runtime support instead of generating bytecode directly. Similarly, expensive arithmetic operations such as unsigned multiplications are also implemented in the runtime layer. Since 32-bit access is easiest to support efficiently, Cibyl focuses on performance for this case.

Cibyl uses a 1-1 mapping between C functions and generated Java methods, which brings a number of benefits. First, this mapping enables the J2ME profiler to produce meaningful output for Cibyl programs. Second, if the program causes an exception, the call chain emitted by the JVM will be human readable. The 1-1 mapping also enables some optimizations, which will be discussed later. We handle register-indirect function calls specially since Java bytecode does not support function pointers. To support function pointers, we generate a special "call table" method that switches on the function address and calls the corresponding method indirectly. Indirect jumps, generated by GCC for example in some switch statements, are handled similarly through a "jump table" in the method which switches on possible jump destinations in the method (which can be found through the ELF relocation information).

Compared to the integer instruction set, the MIPS floating point instruction set is more difficult to translate [8]. Floating point is therefore supported by a hybrid approach where we use the GCC soft-float support, but implement the runtime support functions using native Java floats. GCC generates calls to functions such as `__addsf3` for a floating point add, passing an integer representation of the source registers, and our hybrid approach implements this through a Java method that converts the integer representation to real floats, performs the add and returns the integer-representation of the result. This solution provides a trade-off between implementation complexity and performance, with a very simple implementation but less performance than an implementation of the MIPS FPU instruction set.

3. Optimizations

We perform a number of optimizations in Cibyl apart from the general code generation optimizations described above to improve performance and reduce the size of the generated Java class files.

3.1. 32-bit multiplications/divisions

The MIPS instruction for multiplication always produce a 64-bit result, split in the `hi`/`lo` register pair. We translate this to Java bytecode by casting the source registers to 64-bit longs, perform the multiplication, split the resulting value and place it in `hi`/`lo`. As expected, this generates many instructions in Java bytecode, and is also fairly inefficient and we perform it in the runtime support to save space.

Often, however, only the low 32 bits of the value is actually used and in these cases we perform a 32-bit multiplication which can be done natively in Java bytecode. This is safe since the `lo`/`hi` registers are not communicated across functions in the MIPS ABI, so if the `hi` register is unused we simply skip producing it. This optimization saves both space and improves performance of multiplications. Divisions are handled similarly.

3.2. Size reduction

To reduce size, functions which are unused are pruned during translation, and relocation information in the binary is used to determine which functions are safe to remove. Similarly, the table of indirect calls contains only the functions which are called register-indirect, also determined by relocations and reconstructing values from instructions with relocations (where 32-bit addresses are commonly split between two MIPS instructions).

The 1-1 mapping between C functions and Java methods also allows for size reductions together with the MIPS ABI [13]. On function calls, we pass only the stack pointer and the argument registers used by the called function, which reduces the overhead for short functions. Another optimization the ABI allows is to skip stores and loads in the function prologue and epilogue to the MIPS callee-saved registers `s0`...`s8`, which are handled automatically as the register representation uses Java local variables.

A potential problem with the 1-1 mapping is the 64KB JVM method size limit [9], which puts an effective size limit on each C function. However, from our experience we've rarely seen this problem in practice, with the exception of a the fetch and decode loop of a C64 emulator, which exceeds the limit if the C source file is compiled without optimization.

3.3. Inlining of builtin functionality

We perform selective inlining of certain functions in the runtime support, mostly for the floating point support. This optimization is implemented by matching function call names to a set of Python classes that implements these builtins. These then generate the corresponding functionality through emitting bytecode instructions directly, replacing the function call.

This allows us to reduce the overhead of floating point operations significantly, at the cost of a slightly larger translated file. With this approach, we output Java bytecode for floating point operations directly to inline the GCC helper functions for soft floats. We use this functionality for other purposes as well, e.g., to throw and catch Java exceptions from C code.

3.4. Function co-location

One large source of overhead is method calls, i.e., translated C function calls. Since Cibyl uses local variables for the register representation, it needs to pass up to 7 arguments for the method to call, and method call overhead grows with the number of arguments. This overhead is especially noticeable with short functions which are frequently called.

As a way around this problem, we allow multiple C functions to be co-located into one Java method. This is similar to standard function inlining with the difference that unrelated functions can be co-located in the same method. Calling between functions in a single Java method can then be done using regular `goto`'s avoiding the method call overhead. The implementation is illustrated in Figure 2, which shows a call chain **fn0**, **fn1**, **fn2**:

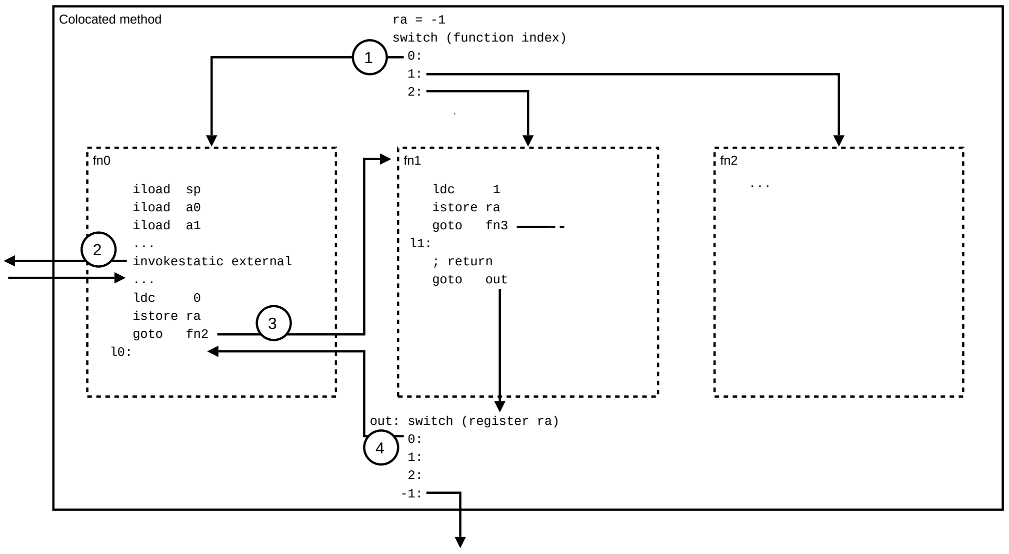
1. Co-located methods are called with an index specifying the function to call, then the method prologue does a switch on this index and calls the function. The MIPS `ra` register is set to a special value to signify that the function was called from outside the method.

2. Calls to external methods are handled as elsewhere, with argument registers (Java local variables) passed.

3. On calls to functions within the method, a direct `goto` is used. Passing argument registers are not needed, but we store a generated index for the return address in the `ra` register (local variable).

4. On returning, we switch on the `ra` register and jump back to where the local call was made or out of the co-located method.

There are a few differences compared to normal methods. First of all we need to save the `ra` register since it's now used for the function returns. Second, we allocate the used MIPS `s`-registers to different Java local variables for each function in the co-located method, which allows us to avoid storing store/restore these registers in the function as with normal methods. The function co-location deliberately relaxes the 1-1 function to method mapping to improve performance.



```
ra = -1
switch (function index)
  0:
  1:
  2:
```

```
fn0
     iload  sp
     iload  a0
     iload  a1
     ...
     invokestatic external
     ...
     ldc     0
     istore ra
     goto   fn2
  l0:
```

```
fn1
    ldc     1
    istore ra
    goto   fn3
l1:
    ; return
    goto   out
```

```
fn2
    ...
```

```
out: switch (register ra)
       0:
       1:
       2:
      -1:
```

Figure 2. Handling of co-located functions in Cibyl

3.5. Constant propagation

We perform constant propagation of the MIPS register values during the compile phase, which improves performance of memory accesses to known addresses. The MIPS instruction set performs all memory accesses with register addresses, and all memory accesses - even those for with known addresses - generate the same code. The MIPS code will therefore look like

```
lui      v0, 0x1000
addiu    v0, v0, 0x100
lw       a0, 0(v0)
```

The implementation keeps a map of current register values, initially set to being unknown. We thereafter execute each instruction and update the register values, generating a constant value if all operands are known (either as known registers or constants). At the start of a basic block, all registers are set to unknown again.

Although constant propagation in general improves code in many places, the executables for Cibyl are already optimized by GCC. The main improvement of this optimization is therefore to direct indexing of loads and stores where the address is known, which is possible in Java bytecode but not with the MIPS instruction set where memory operations always use a register address.

3.6. Peephole optimizer

Cibyl also includes a peephole optimizer to improve common translation patterns. For example, since the size of MIPS instructions is fixed at 4 bytes, constant assignments to registers is split in a `lui`/`addiu` pair to assign the upper and lower half of the register. These assignments are fairly common and are coalesced into a single assignment by the peephole optimizer.

Similarly, storing of intermediate results for computations in registers can often be avoided and kept on the Java operand stack when GCC has generated a temporary computation (discarding the temporary register values). Most patterns which are handled by the peephole optimizer targets cases where MIPS assembly is difficult to translate to Java bytecode.

3.7. Optimization example

Figure 3 shows a sequence of instructions optimized by Cibyl. The instruction sequence comes from a function which shifts and divides one of the input parameters and then prints out the result (the listing ends before the call to printf). The columns in sequence from left to right shows the MIPS instructions, unoptimized Java bytecode instructions, bytecode instructions after constant propagation and 32-bit division optimization and finally the rightmost column shows peephole optimized bytecode instructions.

As can be seen in the non-optimized column, the two difficult cases here are the division and the memory load. The column also shows that the stack store of the return address is nullified which is possible since we use local Java variables. The first optimization step removes the calculation of the division remainder (which is part of the MIPS div instruction) as it is never used. This step also shows

MIPS instructions	Non-optimized	Divisions, constant propagation	Peephole optimized
sll a1, a0, 2	iload_1	iload_1	iload_1
	iconst_2	iconst_2	iconst_2
	ishl	ishl	ishl
	istore_2	istore_2	*dup*
div a1, a0	iload_2	iload_2	iload_2
	iload_1	iload_1	iload_1
	dup2		
	idiv	idiv	idiv
	istore 7	istore 7	istore 7
	irem		
	istore 10		
addiu sp, sp, -24	iinc 0 -24	iinc 0 -24	iinc 0 -24
sw ra, 16(sp)			
lui v0, 0	iconst_0	iconst_0	*aload 6*
	istore 9	istore 9	iconst_0
lw v1, 16(v0)	aload 6	aload 6	istore 9
	iload 9		
	iconst_2		
	iushr		
	iconst_4	iconst_4	iconst_4
	iadd		
	iaload	iaload	iaload
	istore 8	istore 8	istore 8
lui a0, 0	iconst_0	iconst_0	*sipush 256*
	istore_1	istore_1	istore_1
addiu a0, a0, 256	iinc 1 256	iinc 1 256	
mflo a1	iload 7	iload 7	iload 7
	istore_2	istore_2	

Figure 3. Example of Cibyl code optimizations, where the columns show MIPS instructions and different optimization options applied.

the constant propagation which notices that the v0 register is constant and simply pushes the address directly. Data starts at address 0 in Cibyl applications, and this particular word is at address 16 or index 4 into the memory integer array.

The final step is the peephole optimizer, which operates on the Java bytecode generated from the previous steps. There are three transformations performed on this code sequence. First, the store to the local variable 2 (MIPS register `a1`) is pushed forward as it is written again in the `mflo` instruction further down. Assignment to local variable 2 is again is delayed at the `mflo` as in this case it will be overwritten later. Second, we swap the push of the memory vector (`aload 6`) and a constant assignment to simplify other optimizations in later peephole iterations, although in this case there are none. Finally, we coalesce the `lui`/`addiu` pair at the end of the sequence into one assignment.

4. Performance evaluation

We have evaluated Cibyl performance in two benchmarks, FreeMap and A*. FreeMap [15] is a GPS street navigation application originally written for X-windows but later ported to other platforms such as WindowsCE. It has been ported to the J2ME platform using Cibyl by an external developer, and consists of over 40000 lines of C code and 1600 lines of Java code. FreeMap uses a mixture of integer and floating point operations, and displaying the map is the most computationally intensive operation.

We run FreeMap in the Sun J2ME emulator and set it up to perform a map rotation operation during one minute and count the number of iterations of the main loop during this time. Startup and shutdown time is ignored. We compare a non-optimized Cibyl version with an optimized one, where the optimizations enabled are inlining of builtins, optimization of 32-bit multiplications and divisions and co-location of functions dealing with redrawing.

The A* benchmark consists of two implementations of the A* algorithm, one in C and one in Java, which are based on the same source. The Java implementation is not a port, but implemented Java-style using multiple classes. Both implementations stress memory management, dereferencing pointers/references and short function calls. The graph search visits 35004 nodes during the execution.

We setup the A* implementation to use different data types for the main data structure (the nodes in the graph). The types are 32-bit `int`, 16-bit `short`, 32-bit `float` and the 64-bit `double`. We run the benchmark in Cibyl both without and with optimizations, in NestedVM and in

Java on the Sun JDK 1.5.0 [18] JVM, the Kaffe JVM [21], the SableVM [5] interpreter and the GNU Java bytecode interpreter (gij) [20]. The Cibyl optimizations we use is inlining of builtins, memory registers, multiplication and division optimization, and function co-location. We co-locate the functions in the hottest call chain, which is the actual A* algorithm, looking up nodes, iterating over nodes and computing distance heuristics.

All benchmarks are executed on a Pentium M processor at 600 MHz with Debian GNU/Linux. The A* benchmarks are compiled with GCC 3.3.6 (using the NestedVM compiler), using the default options for NestedVM and Cibyl, optimizing at level -O3 in the NestedVM case and -Os, size, for Cibyl. The FreeMap benchmark is compiled with GCC 4.1.2, optimizing for size. The Sun J2ME emulator runs the KVM virtual machine [19] which is also used on many mobile phones.

4.1. Results

For FreeMap, shown in Figure 4, we see that enabling the optimizations improves the performance with almost 15% over the non-optimized version. This improvement is evident in the emulator update frequency, which is visibly smoother with optimizations turned on. The size of the FreeMap classes is 592KB of which 457KB are Cibyl-generated classes and the rest is the runtime support.

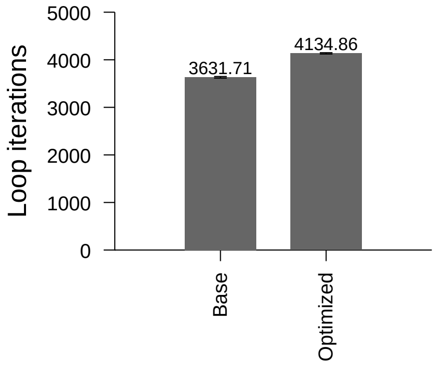


Figure 4. FreeMap benchmark results. The baseline shows results without optimizations enabled.

The largest part of the improvement comes from function co-location and the use of builtins. Both these individually improves the performance with around 13%, and together the improvement reaches 15%. Without these optimizations, the improvement against the unoptimized case is only 4%. The reason builtins and function co-location works well in the FreeMap case is a large amount of floating point operations and short function calls, where the co-location reduces the overhead.

In the A* benchmark, presented in Figure 5, we can see that Cibyl performs very well in the integer-case, being on par with Java on the Sun JVM and significantly faster than NestedVM on all tested JVMs. This is expected since our optimizations target the of 32-bit data case, and the optimizations improve results with between 10-40% depending on JVM. Cibyl is faster than NestedVM also in the unoptimized case, which is most likely caused by the higher use of Java local variables in Cibyl (which are more efficient to access than class members). NestedVM also references memory in a two-level scheme to support sparse address spaces [1], which contributes a bit to the overhead.

For the other cases, there is a more mixed picture. As expected, both Cibyl and NestedVM are far behind Java in these cases since the translated code cannot work directly with 16-bit or 64-bit values and less efficiently with floating point values. With shorts, Cibyl performs better than NestedVM on the Sun JVM and SableVM and marginally worse on Kaffe and Gij. The slowdown compared to the integer benchmark is caused by additional memory latency, and the relative slowdown compared to NestedVM can be explained by the calls into the runtime for 16-bit memory accesses.

The floating point part, both Cibyl and NestedVM frequently need to store floats in integer variables or memory (through the method `Float.floatBitsToInt`), which decrease the performance a lot compared to native Java. For the Sun JVM, Cibyl has performance comparable to NestedVM, but is behind on the other JVMs, which is caused by soft-float overhead. However, we can see that the builtin optimization substantially improves performance with 30-40%. The double case is not optimized by the builtin approach, and show only small improvements from the optimization.

An outlier is native Java on the Kaffe JVM, which shows much worse results than on the other JVMs throughout all the tests. On the other hand, Kaffe gives good results with Cibyl and NestedVM, which both use a simpler program structure based on large static methods.

The size of the classes including runtime support in A* benchmark is 37KB for the optimized Cibyl version, 240KB for NestedVM and 61KB for native Java. Cibyl loads static data (the `.data` and `.rodata` ELF sections) from a file, and with that included in the class size as done in NestedVM and native Java, the Cibyl size is 75KB. The size advantage compared to NestedVM is caused by the larger NestedVM runtime, and the more aggressive use of Java local variables in Cibyl which decreases the size compared to class members.

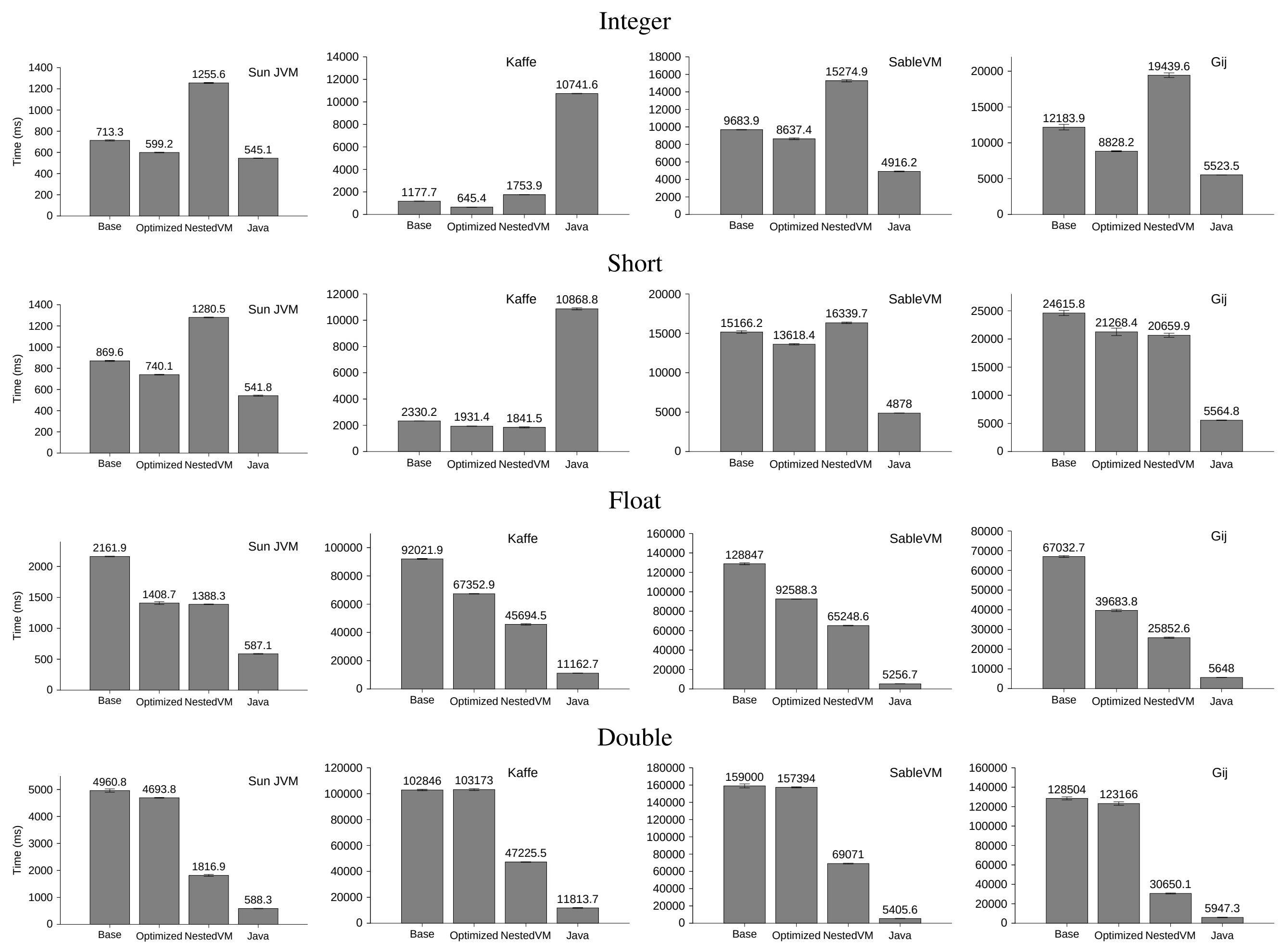


Figure 5. Results of the A* benchmark for the integer, short, float and double data types

5. Related work

NestedVM [1] has many similarities with Cibyl. NestedVM is also a binary translator that translates MIPS binaries into Java bytecode, but NestedVM targets security - being able to run insecure binaries in a secure VM - while Cibyl targets portability to J2ME devices. This is reflected in some of the design decisions, where NestedVM uses a two-level scheme for memory accesses (but which can be disabled) to detect accesses of uninitialized memory and support large address spaces, and an optional UNIX compatibility layer to support translation of existing UNIX tools.

Cibyl on the other hand focuses on generation of compact code and good performance for the common and easily supported case of 32-bit memory accesses. The one-level memory access scheme in Cibyl well suits the embedded J2ME applications we target, where a single application runs at a time and memory availability is limited to a few megabytes. Cibyl also uses Java local variables for the register representation throughout, whereas NestedVM uses local variables only for caching the normal class variable register representation.

There is also a set of compilers which can generate Java bytecode directly. The University of Queensland Binary Translator project has a Java bytecode backend for GCC [4]. This backend is not part of mainline GCC, and tracking mainline development can require a significant effort.

Axiomatic solutions [2] has a compiler for a subset of C which generates Java bytecode. To handle the case of memory references from multiple C files to global memory, the Axiomatic solutions compiler does not support multiple compilation units and requires that all C files are passed to the compiler in one step. The Axiomatic solutions compiler is also based on an old version of GCC. In contrast, Cibyl is independent of GCC version and can therefore leverage improvements to GCC with newer versions. Cibyl also supports the C language fully, and with runtime support any language which GCC can compile (runtime support currently exists for C and C++).

Unfortunately, the non-commercial version of the Axiomatic solutions compiler does not support multiplications or divisions, so the performance tests were not possible to compile with it. A visual inspection of the generated Java bytecode assembly shows that memory references can be generated slightly more efficiently than Cibyl, but also a reliance on placing register values in static Java class variables (also where the values easily could be kept on the operand stack), which is less efficient and more space consuming than using local variables.

6. Conclusions and future work

In this paper, we have presented the optimization framework for the Cibyl binary translator and benchmarked it against NestedVM and native Java. We show how function co-location, constant propagation for register values, inlining of the soft-float implementation and use of the MIPS ABI contributes to improve the performance of code translated with Cibyl. Our benchmarks illustrates how these optimizations can improve performance of real-world Cibyl applications and how binary translation is affected by data types. We also compare Cibyl to the NestedVM binary translator and native Java and show that performance in the case we target is significantly better than NestedVM and close to performance of native Java.

Future directions to improve performance includes implementing the MIPS FPU instruction set, which is an area where NestedVM has an advantage. We are also investigating improved debugging support by using the GDB debugger through Qemu emulator (executing MIPS instructions) and invoking Java functionality through a generated J2ME proxy.

Acknowledgements and Availability

We would like to thank the anonymous reviewers for their valuable input. This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project “Blekinge - Engineering Software Qualities (BESQ)” (`http://www.bth.se/~besq`). Cibyl is free software under the GNU GPL and can be downloaded from `http://cibyl.googlecode.com`.

References

[1] B. Alliet and A. Megacz. Complete translation of unsafe native code to safe bytecode. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators*, pages 32–41, Washington DC, USA, 2004.

[2] Axiomatic solutions. Axiomatic multi-platform C (AMPC). http://www.axiomsol.com/, Accessed 8/9-2006.

[3] F. Buddrus and J. Schödel. Cappuccino - a C++ to java translator. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 660–665, New York, NY, USA, 1998. ACM Press.

[4] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, Mar. 2000.

[5] E. Gagnon. *A portable research framework for the execution of java bytecode*. PhD thesis, McGill University, Montreal, December 2003.

[6] Jazillian, Inc. legacy to ”natural” java translator. see http://www.jazillian.com/index.html, Accessed 8/9-2006.

[7] G. Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[8] S. Kågström, H. Grahn, and L. Lundberg. Cibyl - an environment for language diversity on mobile devices. In *Proceedings of the Virtual Execution Environments (VEE)*, pages 75–81, San Diego, USA, June 2007.

[9] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, Boston, 2nd edition, April 1999.

[10] S. Malabarba, P. Devanbu, and A. Stearns. MoHCA-Java: a tool for C++ to java conversion support. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 650–653, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] J. Martin. *Ephedra - A C to Java Migration Environment*. PhD thesis, University of Victoria, 2002.

[12] J. Meyer and T. Downing. The jasmin assembler. See http://jasmin.sourceforge.net/, Accessed 8/9-2006.

[13] Santa Cruz Operation. *SYSTEM V APPLICATION BINARY INTERFACE - MIPS RISC Processor Supplement*. Santa Cruz Operation, Santa Cruz, USA, 3rd edition, february 1996.

[14] J. Schwartz. Welcome letter, 2006 JavaOne conference. http://java.sun.com/javaone/sf/Jonathans_welcome.jsp, Accessed 8/9-2006.

[15] E. Shabtai. Freemap for J2ME phones. See http://www.freemap.co.il/roadmap_j2me.html/, accessed 2007-09-08.

[16] R. M. Stallman. *Using GCC: The GNU Compiler Collection Reference Manual*. Free Software Foundation, Boston, October 2003.

[17] Sun Microsystems. J2ME. http://java.sun.com/javame/index.jsp, Accessed 8/9-2006.

[18] Sun Microsystems. J2se 5.0. http://java.sun.com/j2se/1.5.0/, Acessed 8/9-2006.

[19] Sun Microsystems. *J2ME Building Blocks for Mobile Devices - Whitepaper on KVM and the Connected, Limited*

Device Configuration CLDC. Sun Microsystems, May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf, Accessed 8/9-2006.

[20] The GNU project. The GNU compiler for the java programming language. http://gcc.gnu.org/java/, Accessed 8/9-2006.

[21] T. Wilkinson, Edouard G. Parmelan, Jim Pick, and et al. The kaffe jvm. http://www.kaffe.org/, Accessed 8/9-2006.