

Cibyl - an Environment for Language Diversity on Mobile 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

With an estimated installation base of around 1 billion units, the Java J2ME platform is one of the largest development targets available. For mobile devices, J2ME is often the only available environment. For the very large body of software written in C other languages, this means difficult and costly porting to another language to support J2ME devices.

This paper presents the Cibyl programming environment which allows existing code written in C and other languages supported by GCC to be recompiled into Java bytecode and run with close to native Java performance on J2ME devices. Cibyl translates compiled MIPS binaries into Java bytecode. In contrast to other approaches, Cibyl supports the full C language, is based on unmodified standard tools, and does not rely on source code conversion. To achieve good performance, Cibyl employs extensions to the MIPS architecture to support low-overhead calls to native Java functionality and use knowledge of the MIPS ABI to avoid computing unused values and transfer unnecessary registers. An evaluation on multiple virtual machines shows that Cibyl achieves performance similar to native Java, with results ranging from a slowdown of around 2 to a speedup of over 9 depending on the JVM and the benchmark.

Categories and Subject Descriptors D.2.6 [Programming Environments]; D.2.7 [Distribution, Maintenance, and Enhancement]; Portability; D.2.12 [Interoperability]

General Terms Languages, Measurement, Performance

Keywords J2ME, Portability, Programming environment, binary translation

1. Introduction

The Java 2 Platform, Microedition (J2ME) [21] has become practically ubiquitous among mobile phones with an estimated installation base of around 1 billion units [18]. J2ME provides a royalty-free development environment where it is possible to extend the capabilities of mobile phones and other embedded systems through Java. J2ME is often the *only* openly available environment for extending mobile phones, and developers writing software to J2ME-capable embedded devices are therefore locked to the Java language. When porting existing software written in languages such

as C or C++ to J2ME devices, the development environment can require a complete rewrite of the software package. Developers are then faced with either porting their code to another language, or use automated tools [5, 10, 13, 14] which may generate code which is difficult to modify, require manual fixes and can sometimes be inefficient. Even when implementing new projects for J2ME, Java might not always be the preferred language. For example, developer language experience, personal preferences, availability of existing libraries or co-development for other targets might favor new implementations in other languages.

In this paper, we present the Cibyl programming environment which allows existing code written in C and other languages to be recompiled as-is or with small modifications into Java bytecode and run on J2ME devices. Performance of the recompiled code can be close to native Java implementations and with modest space overhead. In contrast to other approaches [3], Cibyl supports the full C language, and support for C++ and other languages require only library extensions. Cibyl is not a compiler, but instead relies on the GCC [20] compiler to produce a MIPS binary. Cibyl does a static binary translation of a MIPS executable into Java bytecode, and provides a runtime library to support execution in the Java environment. Compared to writing a backend for GCC which directly generates Java bytecode, the Cibyl approach allows for a lower initial effort and also removes the burden of long-time maintenance of the backend. Using unmodified standard tools also means that it automatically benefits from tool improvements.

The main contributions of the paper are the following. First, we show how C programs can be recompiled into Java bytecode and identify problematic areas. Second, we show that knowledge about the compiled code and the ABI (Application Binary Interface) can be utilized to generate more efficient bytecode. Third, we illustrate how extensions to the MIPS architecture can be used to provide efficient calls to native Java methods.

The rest of the paper is structured as follows. Section 2 describes the technology used in Cibyl. Section 3 presents an evaluation of the generated code in terms of performance and size. Thereafter, Section 4 describes related work, and finally Section 5 presents conclusions and future research.

2. Technology

Cibyl targets the MIPS I [11], only using instructions available in user-space. Compared to many other architectures, MIPS provides a number of advantages for efficient binary translation. First, regular loads and stores are always done on aligned addresses, which simplifies memory handling in Java. Second, MIPS uses the general-purpose register set for almost all operations and does not have implicitly updated flag registers, which allows a straightforward translation of most arithmetic instructions. Third, MIPS only

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'07, June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-630-1/07/0006...\$5.00

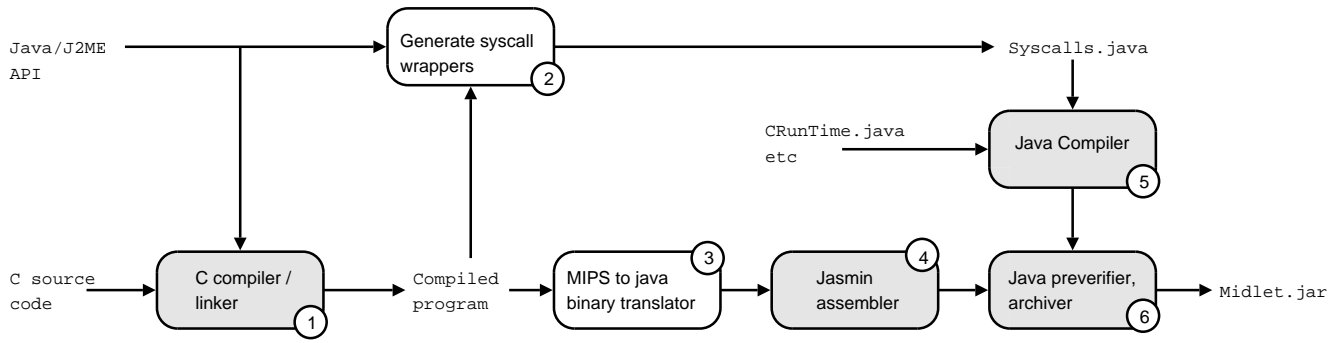


Figure 1. The compilation process. Gray boxes show third-party tools and white boxes are implemented in Cibyl.

does partial register updates for the seldom used unaligned memory accesses instructions.

To achieve good performance of the translated binaries, we place a number of soft restrictions on the generated code and add extensions to the architecture. In particular we focus on good performance of 32-bit memory accesses and operations on signed 32-bit values, which are easier to support efficiently since Java has no unsigned types. We have also made use of extensions to the MIPS ISA, which is possible since the generated code targets a virtual machine and does not need to run on actual hardware.

Cibyl builds on the GNU toolchain [20], which we use to produce the MIPS binaries in a translation-friendly format. GCC is used to compile the C source for the MIPS I instruction set, which is thereafter linked using GNU ld. We use GCC and ld options to simplify certain operations. For example, we turn off the generation of explicit checks for integer division by zero, which is not needed in Java bytecode where the instruction throws a divide-by-zero exception. Further, we always work on static executables and therefore disable the generation of position-independent code. The data and read-only data sections from the ELF binary is placed in a file which the runtime system loads into memory on startup. Cibyl uses five steps to compile C source code into a J2ME JAR-file, illustrated in Figure 1:

1. The C source is compiled and linked with GCC using the Cibyl headers and libraries.
2. The API to Java/J2ME (defined in a C header-file) and the compiled program is passed to another tool that generates a Java source file containing wrappers for system call stubs. The set of system calls used by a program is known at compile time by feeding back the compiled program to the tool and only needed stubs are generated.
3. The `cibyl-mips2java` tool recompiles the linked program from step 2 into Java assembly.
4. The Jasmin [15] assembler compiles the Java assembly into a class file
5. The regular Java compiler compiles the generated system call wrappers and runtime support files
6. Finally, the compiled class-files are preverified and combined by the Java archiver to a downloadable JAR file. The preverification step is needed for J2ME programs since the verification capabilities of the mobile JVM is limited.

2.1 Memory Access

We use a linker script to link the text segment high up in the memory and the initialized and uninitialized data starting at address zero. The translated text segment cannot be addressed, and is there-

fore not loaded into memory. Figure 2 shows the address space in Cibyl. During startup, a configurable portion of the Java heap is allocated to the Cibyl program. The stack pointer is setup to the end of the address space, and the heap starts after the uninitialized data. The heap manager is a standard *malloc/free* implementation written in C.

We strive to provide efficient 32-bit memory accesses while accepting a performance cost for 8- and 16-bit accesses. Memory is therefore represented as an integer-vector, which means that 32-bit loads and stores can be performed by indexing this vector. As the mapped data starts at address 0 and is contiguous from there, the computed address can be used directly as an index after right-shifting it by 2. Figure 3 shows translation of memory accesses.

To further improve the performance of 32-bit memory accesses, we allocate extra registers to optimize multiple memory accesses where the base address register stays the same. The key is that since the base address is constant, the right-shift performed to translate the address into a Java vector index need only be done once. The analysis is done on basic blocks, and replaces registers if there are more than two accesses to a constant address (shown in the right part of Figure 3). With these optimizations, each 32-bit memory access can be done with between 4 and 8 Java bytecode instructions.

8- and 16-bit memory loads and stores also operate on the memory integer-vector, but require more work. For example, a store byte operation must first load the 32-bit word from the memory vector, mask out the requested byte, shift the value to be stored to the correct byte address and perform a bitwise *or* to update the memory location. Signed loads (with the MIPS `lb` and `lh` instructions) also need sign-extension. 8- and 16-bit accesses generate between 20-42 bytecode instructions, depending on the sign and size. To save space, these accesses are performed through functions in the runtime support.

2.2 Code Generation

The MIPS binaries are translated by the `cibyl-mips2java` tool to Java bytecode assembly, which is thereafter assembled into bytecode by the Jasmin assembler [15]. The tool produces one class, which is split up in one method per C function for the recompiled code. During parsing, `nop` instructions and unused functions are discarded and instructions in delay slots are appended to the branch instruction.

We use local Java variables to store registers to improve JVM optimization [6] and produce more compact bytecode. The MIPS `hi` and `lo` registers, which are used to store results from multiplications and divisions, are stored in static variables since these must sometimes be performed in the runtime support. Normal arithmetic instructions require between 2 and 4 bytecode instructions, but we

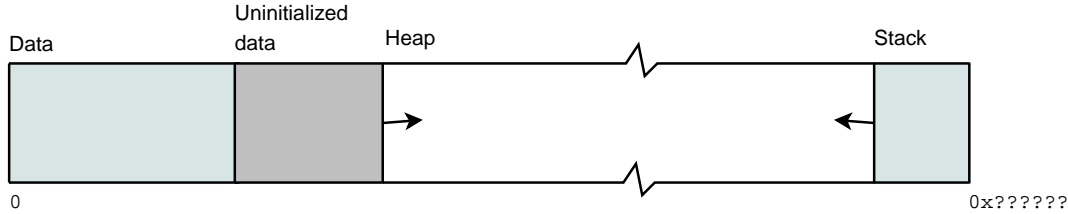


Figure 2. Cibyl address space. The end of the address space depends on the available memory

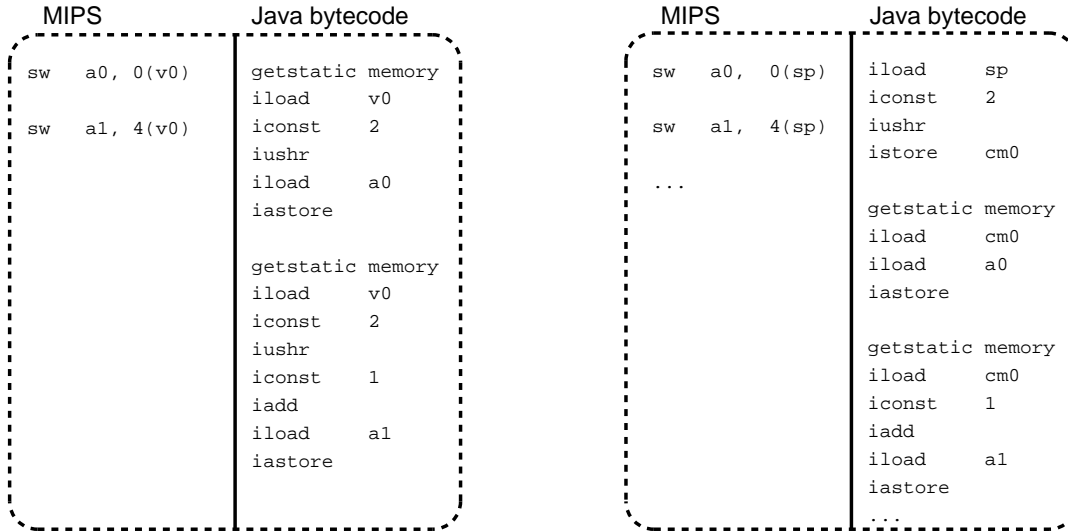


Figure 3. Cibyl memory accesses. The left part shows normal memory accesses and the right part shows memory accesses using the special memory registers.

simplify cases where Java bytecode instructions permit a more efficient translation (e.g., `addi` when used to increase a register by a constant).

We do a number of optimizations on the recompiled code. First, by retaining the relocation information in the MIPS executable, we are able to produce a smaller output binary. The relocation information allows discarding of unused functions (functions which have no entries in the relocation tables), which is not done by the GNU linker. Second, we use architectural extensions to make calls to native Java functionality efficient, which is described in more detail in Section 2.5. Third, by employing knowledge of the MIPS ABI [17] and the program structure, we are able to translate problematic instructions more efficiently.

There are four groups of MIPS instructions that are problematic to translate: instructions for unaligned memory access, instructions dealing with unsigned values, 8- and 16-bit load/stores, and multiplication and division. Unaligned memory access is uncommon in most programs since it is inefficient also on native hardware. We therefore handle these instructions by invoking methods in the runtime environment. Operations on unsigned values are also handled in the runtime where needed, e.g., for multiplications.

Multiplication is problematic because the MIPS `mult` instruction generates a 64-bit result split between the special `hi/lo` registers. In Java, translating `mult` means promotion of the argument to the type `long`, performing a 64-bit multiplication, right-shifting the high part of the result into a temporary and then converting both results back to integers and storing in `hi` and `lo`. The runtime support

for division and multiplication require 9-38 bytecode instructions. For the common case of signed operations where only the low 32-bits are used, we can most of the time perform the operation in the same way as other arithmetic instructions. We do this by omitting the computation of the `hi` value for functions which only read the `lo` value. This can be done because the ABI specifies that function results are never passed in the `hi` or `lo` registers.

A custom peephole optimizer runs on the translated code to remove optimize some inefficiencies in the generated bytecode. This primarily helps with removing extra stores to registers (Java local variables), which are needed in MIPS code but which can be kept on the Java computation stack in Java bytecode.

2.3 Floating point support

Cibyl supports floating point, but does not implement translation of the MIPS floating point unit instructions. Compared to the general-purpose instruction set, the floating point instructions are more difficult to translate efficiently. Many of the floating point instructions have side effects on status registers, and while this can often be handled lazily as done in FX!32 [9], it complicates the implementation. A further problem is that double-precision operations use pairs of single-precision registers, which makes it difficult to store registers in Java local variables and instead requires expensive conversion.

Cibyl supports floating point operations through a hybrid approach where we utilize the GCC software floating point support, but implement it using Java floating point operations, utiliz-

```

typedef union {
    float f; uint32_t i;
} float_union_t;

float __addsf3(float _a, float _b) {
    float_union_t a, b, res;

    a.f = _a; b.f = _b;
    res.i = __addsf3_helper(a.i,b.i);

    return res.f;
}

public static int __addsf3_helper(int _a, int _b) {
    float a = Float.intBitsToFloat(_a);
    float b = Float.intBitsToFloat(_b);

    return Float.floatToIntBits(a + b);
}

```

Figure 4. Cibyl floating point support. The left part of the figure shows the C runtime support, the right part shows the Java implementation of the operation

ing hardware support where available. Figure 4 illustrates how the floating point support works in Cibyl. When compiling for soft-floats, GCC generates calls to runtime support functions for floating point operations, e.g., `__addsf3` to add two `float`'s. The Cibyl implementation of `__addsf3` is shown on the left part of Figure 4, and is simply a call to a Java helper function which is shown on the right part of the figure. The Java implementation will parse the integer pattern as a floating point number (usually just a move to a floating point register), perform the operation and return the resulting integer bit-pattern. This structure requires only runtime support and no changes to the binary translator.

2.4 Function calls

MIPS has two instructions for making procedure calls, `jal` which calls a statically known target address, and `jalr` which makes a register-indirect call. Both these instructions store the return address in the `ra` register. Cibyl maps C functions to static Java methods and makes use of the MIPS ABI [17] to provide better performance and smaller size of the generated code. We disable the GCC optimization of tail calls so that all calls are either done through `jal` or `jalr` instructions. For statically known call targets, Cibyl will then generate a normal Java call to a static method. As an optimization, we only pass registers which are actually used by the function and likewise only return values from functions that modify return registers in the ABI.

Keeping the register state in local variables and a one-to-one mapping of C functions to Java methods provides some benefits. Of the 32 general-purpose MIPS registers, only at most seven are transfer state between functions with the MIPS ABI. These are the stack pointer `sp`, the four argument registers `a0–a3` and the two registers for return values. Other registers are either free to use by the target function or must be preserved. The storing and restoring of the preserved registers in the function prologue and epilogue can be optimized away since each Java method has a private local variable scope.

Since Java bytecode does not allow calls to computed targets, we handle the `jalr` instruction differently. The register-indirect calls are handled through passing the address to a generated static method which looks up the target function address in a lookup table and invokes the corresponding function. While MIPS branch instructions with statically known addresses have corresponding Java bytecode instructions, register-indirect branches (used by GCC for example to optimize switch-statements) pose the same problem as the `jalr` instruction. We also solve this problem in the same way, by a method-local lookup table with possible branch targets in the function. The binary translator use the relocation information and scans the data segment for possible branch targets within the function.

Figure 5 illustrates how indirect function calls work in Cibyl. The code involves two functions, `main` and `printf` where `main` calls `printf` through the register-indirect `jalr` instruction. The indirect is handled via the special `globaltab` method for indirect calls.

2.5 Calls to Native Java Methods

Using extensions to the MIPS architecture, Cibyl allows for efficient invocation of system calls. In most operating systems, system calls are invoked through a register-indexed table and uses fixed registers for arguments. This approach is suboptimal for two reasons. First, the compiler cannot freely schedule registers around the system calls. Second, the invocation is done through a lookup-table, which takes space in the executable and is slower than calling statically known addresses. This effect is aggravated in Java since indirect function calls are not allowed.

We have implemented an efficient scheme to allow native Java functionality to be invoked with close to zero overhead. We achieve this by using special instruction encodings for passing system call arguments and invoking Java methods, which is possible since we are not bound by the restrictions imposed by the pure MIPS instruction set. Technically, the implementation uses the ability of GCC inline assembly to emit register numbers into the generated code.

Figure 6 show the extended MIPS instructions generated for a sequence of instructions which use native Java functionality. Only the return value is fixed to a register (`v0`), otherwise the compiler is able to schedule registers freely. The `get_image` method call uses a constant argument, which is assigned to a temporary register. Since the registers can be chosen freely, the return value of the first method call (in `v0`) is directly used as an argument to the second call (`new_sprite`). The system call invocations are translated into calls of static Java methods in the generated system call wrappers.

2.6 Runtime Support

To support integration with native Java, Cibyl allows passing Java objects to and from C code via integer handles. The runtime environment keeps a registry with mappings between Java objects and handles. Objects are always accessed through the registry.

The C API to access native Java classes is semi-generated, with only the C function prototype being added manually. The C API is structured so that the Java class name and method name can be extracted from the name of the prototype and the Cibyl tools generate accesses to Java objects through the object registry. There are a few cases where the automatic generation of system call wrappers doesn't work, e.g., when passing Java arrays or Java implementations of ANSI C functionality. For these, the Cibyl tools

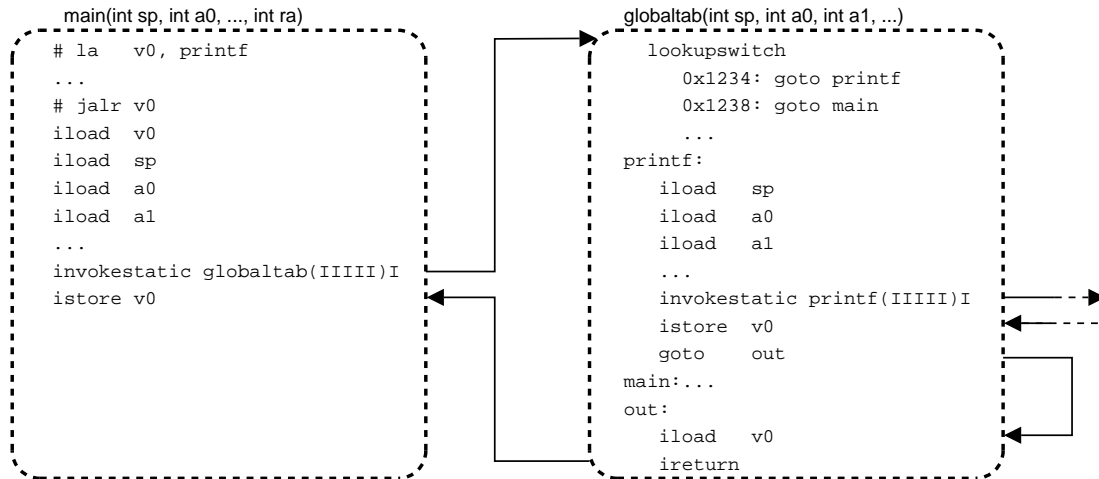


Figure 5. Handling of indirect function calls in Cibyl

```

la          t0, string_address    # image = get_image("/test.png")
syscall_argument  t0
syscall_invoke    35 (get_image)
syscall_argument  v0              # p->sprite = new_sprite(image);
syscall_invoke    20 (new_sprite)
sw              v0,12(a0)

```

Figure 6. System call handling in Cibyl

also support inserting manual implementations of the system call wrappers.

We also implemented a subset of the ANSI C environment with file operations, heap management, most string operations and floating point functions for trigonometry etc. Most of this is implemented in plain C, with helper functions in Java. This is provided as a `libc.a` library file, but since unused functions are pruned it does not add more to the binary size than needed.

3. Evaluation

The Cibyl tools are written in Python, with support libraries for the compiled programs written in C, and the runtime environment in Java. Since we have utilized standard tools whenever possible (e.g., to read and parse ELF files and compile Java assembly to bytecode), the Cibyl tools themselves are fairly small. The tools totally comprise around 3200 lines of code including comments, of which 2600 lines implements the binary translator and the rest implements the generation of system call wrappers and C headers for the system calls.

The runtime support consists of 357 lines of Java code (including comments) and less than 100 lines of C and assembly code (which sets up the environment and calls global constructors). Most of the runtime code implements support for byte and short-sized memory access and the object registry. In addition, the ANSI C environment is currently 1297 lines of C code 368 lines of Java and the soft-float implementation consists of 357 lines of C code and 372 lines of Java.

We have so far ported a number of applications to Cibyl, including several games. For some of these, we have ported the applications to the J2ME C API, and the porting effort then varies depending on how well the API maps to the J2ME API. For others, we have instead left the applications completely untouched and

instead implemented the API in Java as a system call set or in C, using the J2ME API. In most cases, the porting process has been straightforward, mostly consisting of adapting the build system to Cibyl and reimplementing the API-dependent parts for graphics, sound and keyboard input.

The largest Cibyl application we know of is RoadMap [19], a GPS navigation software which was previously available for UNIX and PocketPC platforms. RoadMap uses the Cibyl syscall facilities quite extensively, since the bluetooth GPS support uses an external Java library, and also employs a lot of floating point operations. The RoadMap implementation consists of around 40000 lines of C code and 1300 lines of Java and was completely implemented by an external developer.

3.1 Benchmarks

To see the performance and code size impact compared to native Java we have implemented a number of benchmarks both in Java and in C. The benchmarks are implementations of the Game of Life and the A* algorithms. Both benchmarks are implemented in a Cibyl-friendly way, i.e., using 32-bit values for data accessed in the critical path. We measure the time it takes to run the actual algorithm, disregarding startup time. The benchmarks were executed on a 600MHz Intel Pentium M processor running Debian GNU/Linux. The time is the average of 10 runs.

We also ran two of the benchmarks from the Mediabench benchmark suite [12], ADPCM and PEGWIT. ADPCM is a speech compression benchmark and PEGWIT performs public key encryption and authentication. We limited the selection to these two since many of the benchmarks in the suite uses floating point operations, which is currently stabilizing in Cibyl. These benchmarks compare the native C implementation to the recompiled Cibyl version and we measure only the execution of the actual algorithm (startup time is disregarded).

Benchmark	Lines of code	Cibyl bytes (program/syscalls/C runtime)	Java bytes	MIPS bytes
Life	115	7882 / 3806 / 5394	1853	5204
A*	879	13205 / 3619 / 5394	21897	6836
ADPCM	788	11029 / 4486 / 5394		5572
PEGWIT	7175	83858 / 5313 / 5394		54004

Table 1. The size of compiled classes for Cibyl and native Java, in bytes. The MIPS size is the size of the code segment only. Cibyl size is split in three categories: the program itself, system call wrappers and the C runtime.

JVM (Life)	Cibyl (seconds)	Native Java (seconds)	Slowdown	JVM (A*)	Cibyl (seconds)	Native Java (seconds)	Slowdown
Gij	25.6131	28.5493	0.90	Gij	1.2268	0.6238	1.97
SableVM	20.0237	18.9271	1.06	SableVM	0.9560	0.5320	1.80
Kaffe	2.3426	2.3020	1.02	Kaffe	0.1089	1.0649	0.10
Sun JDK	1.1712	1.3431	0.87	Sun JDK	0.1390	0.1002	1.39

Table 2. Performance results for the A* and game of life benchmarks.

The Game of Life benchmark primarily stresses the memory system with loops of matrix updates. The Java implementation is a straight port from the C implementation, using static methods and static variables for global C variables. We ran the benchmark for 1000 iterations on a 100x100 field. The logic and structure for both A* implementations is the same, but the Java implementation uses multiple classes in the way a native implementation would. The graph search visits 8494 nodes. The A* benchmark stresses function calls, memory allocation and dereferencing of pointers and references.

We have executed the benchmarks with the Sun JDK 1.5.0 [22], the Kaffe JVM [26], the SableVM [8] interpreter and the GNU Java bytecode interpreter (gij) [25]. Gij and SableVM are bytecode interpreters, and therefore similar to the K Virtual Machine [23] common in low-end and older J2ME devices. Kaffe uses a fairly simple Just-in-Time compiler, and is similar to the more recent CLDC HotSpot virtual machine [24]. The Sun JDK has the most advanced virtual machine, which will not be available in J2ME devices in the near future. The C code was compiled with GCC 4.1.2 and optimizes for size with the `-Os` switch. Cibyl optimization was turned on, and the Cibyl peephole optimizer was used to post-process the Java bytecode assembly file.

3.2 Code Size

Table 1 shows the size of the benchmarks for both Cibyl and native Java. The size of the compiled Cibyl programs can be split in three parts: the size of the recompiled program itself, the size of the generated Java wrappers including support classes and the size of the runtime environment. The runtime environment size is constant for all programs, whereas the size of the system call wrappers will depend on the number of system calls referenced by the program. The C library support consumes a significant part of the code for smaller programs, e.g., the `printf` implementation alone consumes 2.5KB in the compiled Cibyl class.

For the A* and game of life benchmarks, we can see that the size of the actual program is within a factor of two of the Java implementation (and for the A* benchmark lower than the Java implementation). Compared to the MIPS code, the size overhead is between 2 and 4 (including the runtime support) for all the benchmarks. For larger programs, we expect the size overhead compared to Java will be small.

3.3 Performance

Table 2 shows the performance results for the A* and game of life benchmarks. The first thing that can be noted is the large perfor-

mance difference between the JVMs, with a 10-30 times performance difference within the same language in the most extreme cases. Secondly, we can see that the performance of Cibyl is within a factor of 2 of the native Java implementation, and in one case clearly outperforms Java.

For the game of life benchmark, GCC was able to optimize the main part of the code very well and this leads to less overhead for Cibyl, which is within 27% of the native Java implementation. A number of interesting properties are shown in the A* benchmark. For all JVMs except Kaffe, this benchmark shows worse results for Cibyl. As with game of life, the two JIT compilers fares better on Cibyl than the pure interpreters. Interesting to note is the extremely good results for Kaffe, which is the fastest result on Cibyl and almost 10 times faster than Java on the same virtual machine, much because of the bad results of the Java implementation.

We believe this is caused by the differences in the generated bytecode. The Java version uses invocations of virtual methods and accesses object fields, whereas Cibyl uses static methods similar to C code. The Kaffe JVM clearly has difficulties with invoking virtual methods and interfaces in the Java implementation, while it optimizes well for the simpler bytecode (static methods) which Cibyl generates.

When comparing recompiled Cibyl code with native C code, there is a large slowdown as shown in Table 3. However, this is mostly because of the current inefficient implementation file operations. For ADPCM, input read with `fread` is the culprit and for PEGWIT, producing the output with `fwrite` causes most of the degradation. By making `fwrite` a no-op, PEGWIT finishes in less than 0.3 seconds, which suggests that improving the performance of file operations should be a future priority.

4. Related Work

NestedVM [1] also performs a binary translation of MIPS binaries to Java bytecode and therefore has many similarities with Cibyl. However, NestedVM has different goals than Cibyl. The main focus of NestedVM is to recompile and run insecure native binaries in a secure VM. In contrast, Cibyl offers an alternative environment on Java-based platforms. NestedVM has a UNIX-compatibility layer to support recompilation and execution of existing UNIX tools, and consequently requires a larger runtime environment.

Technically, NestedVM and Cibyl are also different. To support sparse memory, NestedVM uses a matrix memory representation whereas Cibyl uses a vector. For the embedded applications Cibyl target, the improved performance of the vector representa-

Benchmark	Cibyl (seconds)	Cibyl no file ops (seconds)	Native java (seconds)	Slowdown	Slowdown no file ops.
ADPCM	0.821	N/A	0.031	26.0	N/A
PEGWIT	1.307	0.288	0.051	25.62	5.64

Table 3. Performance results in seconds for the mediabench benchmarks. The Cibyl results were obtained with the Sun JVM.

tion is more important than the ability to support large memories effectively. NestedVM also uses class-variables as register representation whereas Cibyl uses local variables, which gives more efficient and compact bytecode. The use of architectural extensions also separates Cibyl from NestedVM, and Cibyl uses a hybrid software floating point implementation while NestedVM implements the MIPS FPU instructions.

There are also a few compilers which generate Java bytecode directly. Axiomatic solutions [3] has a compiler for a subset of C which generates Java bytecode, and the University of Queensland Binary Translator project has developed a Java bytecode backend for GCC [7]. Compared to the Axiomatic solutions compiler, Cibyl provides full support for the C language can leverage the GCC optimization framework. The Java bytecode backend is not part of GCC and therefore requires a significant effort to track mainline development. In contrast, maintenance of Cibyl is independent of GCC and benefits automatically from GCC updates and Cibyl also provides a complete development environment for J2ME devices.

The area of binary translation can roughly be separated into two areas: static and dynamic binary translators, with Cibyl being a static binary translator. A cited problem with static translators is separating code from data [2] but being a development environment, Cibyl does not have these problems. By retaining relocation and symbol information from the compiler and using static linking, Cibyl cleanly separates code from data and can prune unused functions and executing data or rewriting code is not possible in Java. Dynamic binary translators, performing the translation during runtime, avoid problems with static binary translation, but typically targets other problems than Cibyl such as translating unmodified binaries from one architecture to another [9], performing otherwise difficult optimizations [4] or implementing debug and inspection support [16]. Since Cibyl targets memory-constrained embedded systems, the large runtime support system needed for a dynamic translator would be a disadvantage.

5. Conclusions

In this paper, we have presented the Cibyl binary translation environment. We have described how extensions to the MIPS architecture and use of the ABI can help bring the performance close to that of native Java implementations. We have further described how Java functionality can be integrated into C programs in the Cibyl environment efficiently and with small effort. The approach taken by Cibyl should be comparable in performance to a dedicated compiler backend for Java bytecode, with much less effort. We believe that Cibyl can fill an important niche for porting of existing programs in C and other languages to the J2ME platform, but also for development of new projects where Java might not be the ideal language. We also expect that the small size of the Cibyl tools make it easy to maintain in the long run. Since we use unmodified standard tools, Cibyl benefits from improvements and new tool versions without the need to continuously track development.

There are multiple possible directions for future research on Cibyl. First, to reduce the overhead of function calls, functions which are called in a chain (determined through profiling) can be colocated to one Java method with a common entry point. Second, to further improve performance and reduce size, we are planning an implementation of register value tracking. Third, to better support

debugging, we are investigating an implementation of GDB support for Cibyl. Fourth, runtime libraries for C++ and other languages would further increase the applicability of Cibyl.

Acknowledgements and Availability

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://spel.bth.se/index.php/Cibyl>.

References

- [1] ALLIET, B., AND MEGACZ, A. Complete translation of unsafe native code to safe bytecode. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators* (Washington DC, USA, 2004), pp. 32–41.
- [2] ALTMAN, E., KAELI, D., AND SHEFFER, Y. Welcome to the opportunities of binary translation. *Computer* (March 2000).
- [3] AXIOMATIC SOLUTIONS. Axiomatic multi-platform C (AMPC). <http://www.axiomsol.com/>, Accessed 8/9-2006.
- [4] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices* 35, 5 (2000), 1–12.
- [5] BUDDRUS, F., AND SCHÖDEL, J. Cappuccino - a C++ to java translator. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing* (New York, NY, USA, 1998), ACM Press, pp. 660–665.
- [6] BUDIMLIC, Z., AND KENNEDY, K. Optimizing Java: theory and practice. *Concurrency: Practice and Experience* 9, 6 (1997), 445–463.
- [7] CIFUENTES, C., AND EMMERIK, M. V. UQBT: Adaptable binary translation at low cost. *IEEE Computer* 33, 3 (Mar. 2000), 60–66.
- [8] GAGNON, E. *A portable research framework for the execution of java bytecode*. PhD thesis, McGill University, Montreal, December 2003.
- [9] HOOKWAY, R. J., AND HERDEG, M. A. Digital FX!32: combining emulation and binary translation. *Digital Technology Journal* 9, 1 (1997), 3–12.
- [10] JAZILLIAN, INC. legacy to “natural” java translator. see <http://www.jazillian.com/index.html>, Accessed 8/9-2006.
- [11] KANE, G. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [12] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture* (1997), pp. 330–335.
- [13] MALABARBA, S., DEVANBU, P., AND STEARNS, A. MoHCA-Java: a tool for C++ to java conversion support. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 650–653.
- [14] MARTIN, J. *Ephedra - A C to Java Migration Environment*. PhD thesis, University of Victoria, 2002.
- [15] MEYER, J., AND DOWNING, T. The jasmin assembler. See <http://jasmin.sourceforge.net/>, Accessed 8/9-2006.

- [16] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).
- [17] SANTA CRUZ OPERATION. *SYSTEM V APPLICATION BINARY INTERFACE - MIPS RISC Processor Supplement*, 3rd ed. Santa Cruz Operation, Santa Cruz, USA, february 1996.
- [18] SCHWARTZ, J. Welcome letter, 2006 JavaOne conference. http://java.sun.com/javaone/sf/Jonathans_welcome.jsp, Accessed 8/9-2006.
- [19] SHABTAI, E. RoadMap for J2ME phones. <http://www.freemap.co.il/roadmap/>, accessed 2007-03-27.
- [20] STALLMAN, R. M. *Using GCC: The GNU Compiler Collection Reference Manual*. Free Software Foundation, Boston, October 2003.
- [21] SUN MICROSYSTEMS. J2ME. <http://java.sun.com/javame/index.jsp>, Accessed 8/9-2006.
- [22] SUN MICROSYSTEMS. J2se 5.0. <http://java.sun.com/j2se/1.5.0/>, Accessed 8/9-2006.
- [23] 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.
- [24] SUN MICROSYSTEMS. *The CLDC HotSpot Implementation Virtual Machine*. Sun Microsystems, February 2005. http://java.sun.com/~j2me/docs/pdf/CLDC-HI.whitepaper-February_2005.pdf, Accessed 8/9-2006.
- [25] THE GNU PROJECT. The GNU compiler for the java programming language. <http://gcc.gnu.org/java/>, Accessed 8/9-2006.
- [26] WILKINSON, T., EDOUARD G. PARMELAN, JIM PICK, AND ET AL. The kaffe jvm. <http://www.kaffe.org/>, Accessed 8/9-2006.