Implementing a Portable Foreign Function Interface through Dynamic C Code Generation

Abstract

Interpreters for various languages usually need to interface to arbitrary library functions that have a C interface. Unfortunately, C does not directly allow constructing calls to functions with arbitrary parameter lists at run-time. Various libraries (in particular libffi and the ffcall libraries) have been developed to close this gap, but they have two disadvantages: each foreign function call is slow, and these libraries are not available on all platforms. In this paper we present a method for performing such foreign function calls by creating and compiling wrapper functions in C at run time, and then dynamically linking them. The technique is portable to all platforms that support dynamic linking. Moreover, the wrapper functions are faster than foreign function calls through earlier libraries, resulting in a speedup by a factor of 1.28 over libffi on the DaCapo Jython benchmark. One disadvantage is the higher startup cost from calling the C compiler at run time. We present and evaluate several methods to alleviate this problem: caching, batching, and seeding.

1 Introduction

Many languages support calling libraries written in other programming languages; this facility is known as foreign function interface. More concretely, many of the libraries of interest provide C interfaces, so the foreign language of interest is usually C. Foreign function interfaces are particularly popular in scripting languages such as Python [Bea98], Perl, Tcl, and Ruby, but are also present in other languages, e.g., Java [Lia99], Common Lisp, LuaML[Ram03], Haskell [FLMJ98], and various Forth implementations.

Interpreters written in C are a popular implementation technique for programming languages, in particular scripting languages. They combine a number of advantages:

- simplicity
- portability across architectures
- high startup speed

One might think that calling C functions from an interpreter written in C should be trivial, but unfortunately in the general case this is not true: One can call specific C functions easily; one can also call C functions with a specific sequence of argument and return types indirectly; but calling a function with a statically unknown sequence of argument and return types is not supported by C.

The ffcall libraries[1] and libffi[2] have been developed to fill that gap, but they introduce a significant call overhead and they are not available on all platforms.

In this paper we present and evaluate a foreign function interface based on generating wrapper functions in C at run-time, and then compiling and dynamically linking these functions. Some may consider this idea obvious, but few people have used it (and even fewer have published about it), people have invested significant effort in other approaches (e.g., libffi and ffcall), and there have been no empirical evaluations of this approach; the present paper fills this hole.

Our main contributions in this paper are:

- We present an implementation of this idea in a JVM interpreter (Section 3.2) and compare the calling performance with libffi and ffcall (Section 5.2).

[1] http://www.haible.de/bruno/packages-ffcall.html
[2] http://gcc.gnu.org/viewcvs/trunk/libffi/

- We also discuss the run-time cost of invoking the C compiler (Section 4) and present and evaluate several ways of alleviating this problem:
  - Using a faster compiler (Section 4.1, Section 5.4)
  - Caching of compiled wrapper functions between runs (Section 4.2, Section 5.3)
  - Compiling a batch of wrapper functions at once (Section 4.3, Section 5.4)
  - Seeding the cache with wrapper functions for popular foreign functions (Section 4.4)

We discuss the problem and its existing solutions in more depth in Section 2, and finally compare our approach with related work (Section 6).

2 Previous work

In this section we describe using the ffcall libraries and libffi for calling foreign functions, and the disadvantages of this approach.

You can use the ffcall interface for calling a foreign function (avcall) as follows: For each parameter, you call a type-specific macro (e.g., `av_int`), and pass the actual parameter. These macros construct an `av_list` data structure, that you eventually use in the `av_call` macro to call the function. The overhead of using this interface for a foreign function call in an interpreter consists of several components:

- The interpreter has to interpret a description of the foreign function, and use that to call the appropriate ffcall macros in the appropriate order.
- The macros construct a data structure.
- Finally, `av_call` has to interpret that data structure, then move the parameters to the right locations according to the calling convention, then perform the call, and finally store the result in the right location.

It is easy to see that this is quite a bit more expensive than a regular function call. However, we were still surprised when we saw significant speedups from our new method not just on microbenchmarks, but also on application benchmarks.

The libffi interface is similar, but divides the work into two stages: In the first stage you pass an array of parameter types to the macro `ffi_prep_cif`, and libffi creates a call interface object from that. In the second stage you create an array of pointers to the parameter values, and pass that to the calling macro `ffi_call` along with the call interface object.

In the Cacao interpreter, we execute the first stage only once per foreign function, and the second stage on every call. In theory, this two-stage arrangement can be more efficient than the ffcall interface, but in practice, the ffcall interface is faster on the platforms where we measured both.[3] But even if the libffi interface was implemented optimally, there would still be at least the overhead of constructing the array of parameter pointers.

Another issue with the approach taken by these libraries is that they require additional effort for every architecture and calling convention. While the maintainers of these libraries have done an admirable job at supporting a wide variety of architectures and calling conventions, they are not complete: e.g., at the time of writing the Win64 calling convention for the x86-64 (aka x64) architecture is supported by neither library. So, an approach that does not require extra work for every architecture and calling convention would be preferable for completeness as well as for reducing the total amount of effort.

3 Dynamically Generated C Wrapper Functions

In this section we describe the basic idea and two implementations of our approach. We present improvements and optimizations beyond the basic approach in Section 4, and timing results in Section 5.

3.1 Basic Idea

The C compiler already knows the architecture and the calling convention of the platform, so we should

[3] In our experience, libffi still has one significant practical advantage over ffcall for interpreter developers: gdb backtracing works across libffi calls, but not across ffcalls.

make use of this knowledge instead of reimplementing it from scratch (as is done in ffcall and libffi).

Unfortunately, the C language does not give us a way to use this knowledge directly, e.g., through a run-time code generation interface. But we can perform run-time code generation indirectly: generate C source code at run-time, then invoke the C compiler at run-time, and finally link the resulting object file dynamically.

What is the C code that we need to generate dynamically if we want to call arbitrary functions? It needs to be a C function, and we will call this the wrapper function. It must be callable from ordinary C code, so it needs pre-determined parameter and return types, typically the same for all wrapper functions. The wrapper function needs to call the foreign function, and pass the parameters to it. It has to read these parameters from the locations where the interpreter has put them; in a stack-based virtual machine (VM), it will typically read the parameters from the VM stack. And the wrapper function has to store the result of the foreign function in a place where the interpreter expects it; again, this will usually be the stack in a stack-based VM. The wrapper may also update the VM state (in particular, the VM stack pointer). Figure 1 shows and Section 3.2 explains an example of such a wrapper function.

After dynamic linking, the wrapper function can be called arbitrarily often with an ordinary indirect call in C.

3.2 Implementation in a JVM

We have implemented this idea in the Cacao interpreter [EGKP02, ETK06], a JVM implementation. It is used for calling Java Native Interface (JNI) functions. The wrapper functions are generated, compiled, linked, and run in a just-in-time fashion, i.e., when the JNI function is first called.

Figure 1 shows an example of a wrapper function for calling the JNI function Java_java_io_VMFile_list. All wrapper functions take the JVM stack pointer as parameter and return the updated stack pointer as return value. The JNI function has three parameters:

1. As for all JNI functions, the first parameter is the `JNIEnv` pointer (declared as `void` pointer to avoid having to include `.h` files).

2. The function is for a static class method, so the second parameter has to be the class pointer. We pass this pointer through a variable, and assign the class pointer to the variable when we dynamically link the wrapper function (and the variable). We have one such variable per static class function; sharing the variables for the same class would be possible but more complex. In an earlier version we passed the class pointer as a literal number, but we changed it to the current scheme in order to implement caching (Section 4.2), because different runs may require different literal addresses for the same class (e.g., due to address-space randomization).

3. The rest of the parameters are the parameters passed to the method at the JVM level. In this example the called function has one pointer parameter. The parameters reside on the JVM stack and are accessed through the stack pointer. The offsets from the stack pointer (0 in this case) are hard-coded into the wrapper function. The type of the parameter is also hard-coded, in the form of a cast of the pointer type before the access; in the example, the type of the parameter is a pointer, so there is a cast to `void **` before the access. The types and offsets of the parameters are computed from the descriptor of the method.

The wrapper function stores the result of the JNI function back to the stack, again using a hard-coded type and offset derived from the method descriptor. Finally, the wrapper function returns an updated stack pointer, again with a hard-coded increment.

4 Improvements

This section discusses improvements over the basic approach that address the following problems:

The main disadvantage of our approach is that invoking the C compiler is an expensive operation, and invoking it a lot increases the startup time of the programs significantly. Another disadvantage in some settings (in particular, embedded systems) is that our approach requires a C compiler and toolchain to be available at run-time.

```
extern void *_Jv_env;
void *ccjni_class_Java_java_io_VMFile_list;
long *ccjni_Java_java_io_VMFile_list(long *sp)
{
  extern void *Java_java_io_VMFile_list();
  *(void **)(sp+0) = Java_java_io_VMFile_list(_Jv_env,
          ccjni_class_Java_java_io_VMFile_list, *(void **)(sp+0));
  return sp+0;
}
```

Figure 1: A wrapper function for a JNI function

4.1 Faster compiler

A straightforward improvement with respect to startup time is to use a faster compiler; a very fast one, possibly the fastest, is tcc[4]. However, compilers often pay for faster compilation with slower execution. Moreover, even a very fast C compiler still requires a significant startup time: E.g., compiling 488 files, each with one wrapper function, with 488 invocations of tcc 0.9.22 requires 1.1s on a 2.26GHz Pentium 4. Finally, fast C compilers are not generally available: tcc supports only a few platforms, and is often not even installed on those platforms where it is available. Therefore additional ways to improve the compilation time would be useful.

4.2 Caching

Instead of generating and compiling the same wrapper function every time we invoke a given foreign function, we can just keep the shared object file for the wrapper function around and reuse it in the next run of any program that uses that foreign function.

We have implemented this approach in the Cacao interpreter as follows: the file name of the shared object is derived from the name of the JNI function, so we just need to check if there is an appropriately named file; only if not, generate and compile a wrapper function file. If the file already exists, the startup cost for this JNI function is reduced to almost nothing.

There are a number of issues that have to be dealt with when implementing caching:

Race conditions. If several instances of the interpreter execute at the same time, they have to avoid writing and compiling wrapper function files at the same time. This problem can be avoided with appropriate locking.

Security. An interpreter run by user Alice must not dynamically link shared object files belonging to a different (untrusted) user Bob, because (with a suitably constructed shared object file) that would give Bob all the privileges of Alice. This can be avoided by keeping per-user caches, which would typically reside at a subdirectory of the user's home directory. Also, the foreign function interface should check that that subdirectory and the cache files belong to the user.

Proper architecture. The shared object files must be compiled for the architecture and ABI that the interpreter is running on. The user's home directory can reside on a network file system and be mounted on machines with different architectures, so a particular shared object file might have been created by the interpreter on a different architecture. This problem can be avoided by having per-architecture directories for the shared-object files.

Caching works very well. However, to reduce the startup overhead on the first execution, one can use the following techniques.

4.3 Batching

We can collect several wrapper functions into one file, and compile (and dynamically link) all of them

[4] http://fabrice.bellard.free.fr/tcc/

in one batch. That eliminates much of the repeated startup overhead of the C compiler; other (minor) benefits of batching are that there are fewer shared object files that have to be linked dynamically, resulting in less load on the virtual memory system, and reduced memory consumption from internal fragmentation in the pages.

We have not implemented batching in the Cacao interpreter, because that would require substantial changes in the way Cacao deals with JNI functions; currently Cacao only deals with JNI functions in a JIT way, i.e., at the execution of first call to the function; at that point the wrapper function has to be compiled immediately, leaving no time for accumulating more wrapper functions.

One way to achieve batching in a JVM would be to generate a file of wrapper functions when loading a class containing JNI functions. One might even accumulate a file containing wrapper functions for several classes, and only compile all of them when one of the functions is actually called.

Combining batching and caching complicates things a little: We can no longer use the file name to determine if a wrapper function is cached, but need an index file or data base that indicates which wrapper function is in which file.

4.4 Seeding

Wrapper functions for popular foreign functions can be generated and compiled when the interpreter is built, and later the system can use the resulting shared object files without having to generate them itself, reducing the first-execution slowdown that the user experiences. Another benefit of this approach is that these shared objects would not have to be replicated for each user.

This approach can also be useful to deal with platforms where no C compiler is available at runtime (e.g., embedded systems): Seed the system with shared objects for all wrapper functions that the application needs, and it will never invoke the C compiler. While this approach will not work for every application, many applications on embedded systems are so well-characterized that all the needed foreign functions are known in advance.

4.5 Call Templates

We have considered and others have suggested to solve the foreign function call problem as follows: Instead of having calls to concrete functions with their parameter and return types, provide a set of indirect calls to C functions with various combinations of parameter and return types; when you want to call a given C function, execute the appropriate indirect call with the function pointer as parameter.

The problem with this approach for foreign function calls is that it cannot be complete, because one cannot provide enough call templates to cover all possible parameter and return types. E.g., the ffcall library avcall supports 14 different parameter types (plus `void` for return types). With only three parameters or less, 44325 call templates would be needed.

But these templates are not enough: one wants to be able to call functions with more parameters, and in some foreign function interfaces, one also wants to deal with more types[5], resulting in an even larger number of call templates. That's why we did not implement this approach, and we are not aware of anyone who has used it for implementing a foreign function interface.

However, call templates could be used for reducing the amount of compilation needed, similar to seeding: Provide call templates for the type signatures of the expected foreign functions. Then most of the other foreign functions will also be callable by these templates, and only a few foreign functions need a newly-compiled wrapper function (or, alternatively, a newly-compiled call template).

If the call templates are implemented as wrapper functions, using them would be slightly slower than using the wrappers our approach generates: The call to the foreign function would be indirect, and the called function has to be passed as a parameter to the wrapper function.

Alternatively, for a limited number of call templates, one could implement the call directly in the interpreter, resulting in a small speedup compared to our wrapper function approach: one level of call overhead would be eliminated.

[5] E.g., the mapping of the C type `off_t` to the basic types provided by ffcall is platform-dependent, so for a more portable foreign function interface one may want to provide an `off_t` type.

We have not implemented either variant of call templates, but we expect the calling-performance difference from our direct-calling wrapper functions to be small (probably in the measurement noise), and the startup-time advantage to be similar to seeding or having a warm cache.

The downside of this approach is that it increases the complexity of the implementation: Generating wrapper functions is still needed to support the general case, but in addition one has to map the called functions to the call templates, and to support another way of calling foreign functions.

5 Empirical Results

5.1 Platform and Benchmarks

Unless otherwise noted, our empirical results were gathered on an Intel Xeon UP 3070 2.67GHz machine with 8 GB of main memory running on 64-bit Debian GNU/Linux 4.0 with the Linux kernel version 2.6.18, libffi version 4.1.1-21, ffcall version 1.10+2.41-3, and Cacao based on version 0.97 built with gcc version 4.1.1-21. The benchmarks we used are SPECjvm98 [SPEC99], the DaCapo benchmark suite [MMB+04] and some micro-benchmarks.

5.2 Calling speed

To compare the raw call performance of the different approaches, we wrote a microbenchmark that invokes an empty static JNI method with n arguments (n+2 arguments for the C function) a million times. Figure 2 shows the time for a single call in CPU cyles. While libffi with its two-stage interface could be faster than ffcall, it is quite a lot slower on our test platform. Using C wrapper functions beats both of them, with a speedup of 1.18–1.47 over ffcall and 2.68–7.17 over libffi.

However, these are best-case results that do not take into account that the functions or the surrounding Java program takes time for doing something useful, nor does it include the compilation or dynamic-linking overhead. So, how well do the different approaches work on application benchmarks?

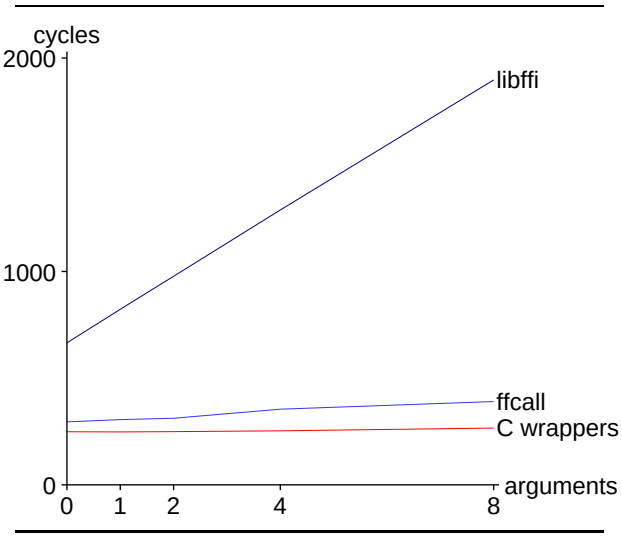

Figure 2: Calling cost of a static JNI call using different foreign function call implementations.

5.3 Application benchmarks, caching and seeding

In our application benchmarks we show two results for our C wrapper approach:

no caching This is the performance of C wrappers without caching, or with a cold cache (i.e. on the first execution of the benchmark after clearing the cache[6]). Each wrapper function is generated, compiled, and dynamically-linked separately; there is one wrapper function per JNI function.

warm cache This is the performance of C wrappers with caching, when all the wrapper functions have already been generated, i.e., typically on the second and subsequent runs of the benchmark. There is no generation or compilation of the wrapper functions, only the dynamic-linking cost, once per JNI function. This is also the performance that we can expect on first execution from seeding if all JNI functions used in the application are seeded.

Figure 3 compares the performance of the foreign function interface implementations on the JVM98

[6]If the cache has not been cleared, an application typically benefits quite a lot from wrappers for JNI functions called by other applications.

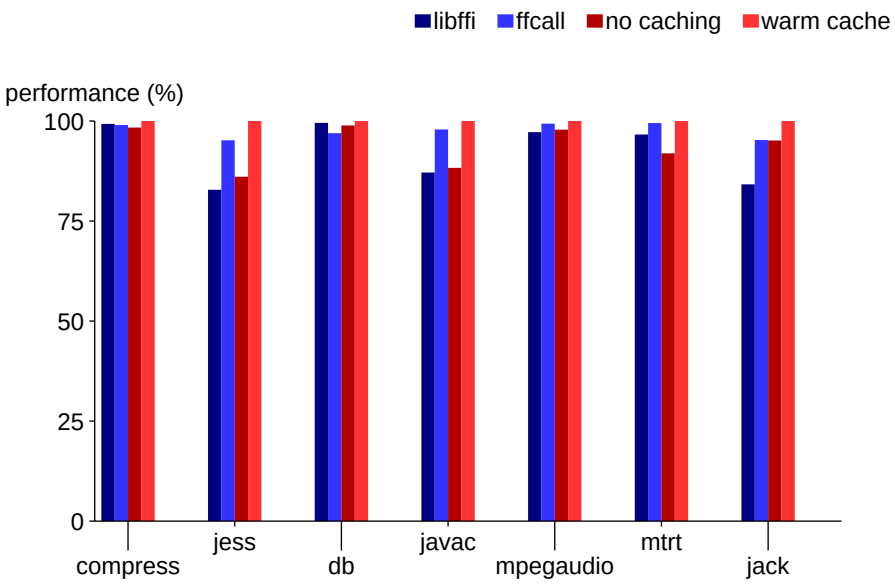

Figure 3: SPECjvm98 results (128MB Java heap)

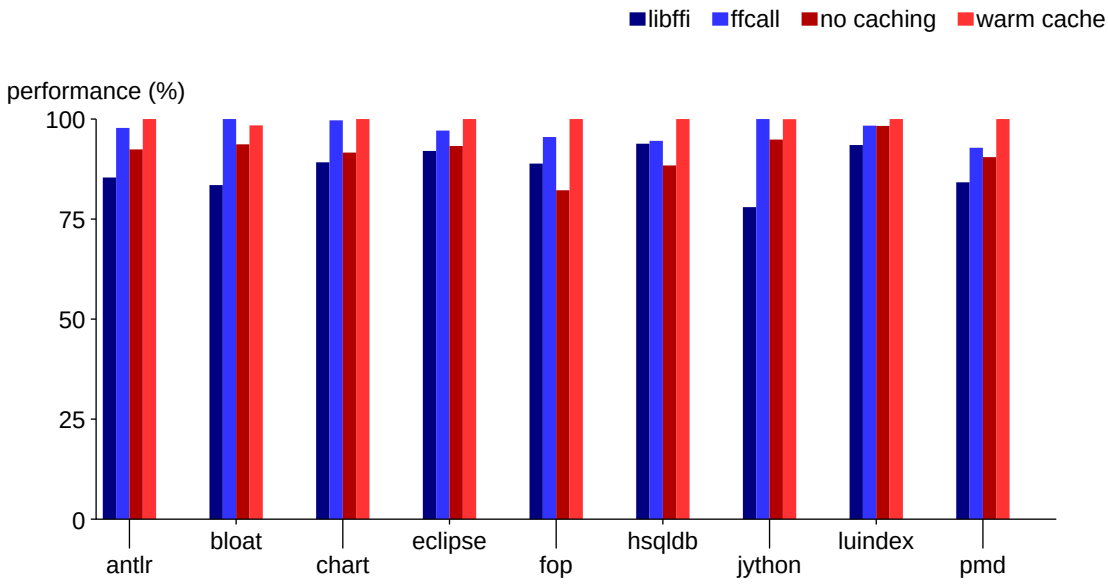

Figure 4: DaCapo benchmark results (512MB Java heap)

	different JNI calls	dynamic JNI invocations
_201_compress	66	4,863
_202_jess	79	2,708,609
_209_db	67	127,294
_213_javac	81	4,296,649
_222_mpegaudio	75	1,253,804
_227_mtrt	78	251,370
_228_jack	68	2,825,405
antlr	91	4,311,262
bloat	119	18,941,773
chart	149	13,026,159
eclipse	321	29,320,684
fop	99	1,871,976
hsqldb	116	449,259
jython	136	49,891,342
luindex	102	7,037,744
pmd	105	12,908,311

Figure 5: Number of JNI methods and calls for the JVM98 and DaCapo benchmarks

benchmarks, and Fig. 4 for the DaCapo benchmarks.

The results exceeded our expectations by far. Not only did we see measurable speedups of *warm cache* over both libffi and ffcall, in several cases even *no caching* was faster than libffi. The maximum speedup we saw was a factor of 1.28 for *warm cache* over libffi on jython. These results are surprising because we did not expect our interpreter to spend that much time in JNI functions, much less in the function call overhead for these functions.

One contributing factor to these results is the different cost of the different calling mechanisms as discussed in Section 5.2. The other contributing factor is the large number of JNI calls, shown in Fig. 5. E.g., for Jython on average each libffi call costs 220ns more than each C wrapper call, resulting in an overall execution time difference of 11s, exceeding the time to compile the 136 wrapper functions in the *no caching* case.

There are also a few unexpected results: e.g., in Jython ffcall is just as fast as *warm cache*, and in db libffi is faster than ffcall. We believe that these differences are due to factors like cache effects (e.g., the difference in allocations between the variants can lead to different conflict misses).

	libffi	ffcall	warm cache	static seeding
runtime	10ms	9ms	43ms	1ms

Figure 6: Dynamic linking overhead: CPU time for calling 500 static JNI methods

	gcc-3.3.5	tcc-0.9.22
1 compile/function	9108ms	1077ms
1 compile/488 functions	1065ms	35ms
functions for break-even	9	33

Figure 7: CPU (user+system) time for compiling 488 wrapper functions on a 2.26GHz Pentium 4 (32-bit)

Figure 6 gives an idea of the dynamic linking cost for warm caches: It measures the CPU (user+system) time for calling 500 different static JNI functions with no arguments, calling each function just once. Here *warm cache* is quite a bit slower than libffi and ffcall, because it needs to dynamically load the 500 additional shared libraries and dynamically link the 500 wrapper functions (in addition to the 500 JNI functions, which are also linked by libffi and ffcall); however, the absolute cost of this dynamic linking overhead is small.

Figure 6 also contains a column *static seeding* where the wrapper functions have already been linked statically. This eliminates the dynamic linking overhead, resulting in a very low run-time for this micro-benchmark.

5.4 Batching

We cannot provide application benchmarks results with batching, because we have not implemented batching in the Cacao interpreter.

However, Fig. 7 shows the potential of batching: gcc -O shows more than a factor of 8 speedup from compiling all the wrapper functions in one batch; the tcc speedup is a factor of 30, and the resulting compile time is not human-perceptible at 35ms.

The third line shows how many functions need to be compiled in one invocation to make the actual compilation time at least as large as the startup overhead of the compiler; this gives an idea of a kind of break-even point for the compiler.

For the runs with one compiler invocation per

function, the system time consumed a large part of the CPU (for tcc, the majority), probably because the startup overhead involves a lot of virtual memory work, in particular page table setup and zeroing or copying of pages on writing. The operating system for this experiment was Linux-2.6.13.

6 Related Work

The ffcall libraries and libffi achieve a similar goal as our approach does, but the approach is different: We use the C compiler's knowledge of the calling convention to achieve portability, these libraries contain hand-written code for each architecture and calling convention they support. You can read a description of how they are used and their drawbacks in Section 2.

Generating wrapper functions to implement a foreign function is already introduced in an earlier paper [Ert07]. However, that paper differs from the present one in all other respects. In particular, it does not present any performance results, so it is unclear how practical this approach is; it does not cover optimisations such as caching, batching, or seeding. It also does not discuss Java or JNI at all. Instead, it deals with Forth and focusses on the design of the foreign function interface as seen by the Forth programmer. Using wrapper functions compiled by the C compiler provides additional advantages in this context, because one can make good use of the C compiler's knowledge about the type signature of the called function.

There is a lot of other work on wrapper and interface generation. The main difference from our work is that wrapper and interface generation is not done at run time of the program.

SWIG [Bea96, Bea03] is a tool for generating foreign function interfaces for scripting languages. It is highly configurable and can be used to generate C wrapper files similar to the ones we use; however, the typical use is quite far from what we have presented in this paper (but closer to our work on the Gforth foreign function interface [Ert07]); in the present paper we have hardly looked at dealing with the mismatch between two languages, because this has been defined away with JNI (for the JVM) and for Gforth has been covered in another paper [Ert07]. SWIG is normally not invoked at run-time, but at build time, whereas we focus on C code generation at run-time. SWIG uses C and C++ as interface definition language (IDL) and is one of most portable systems. TIDE is another foreign function interface generator for interfacing Tcl with full C++ [Win97].

Other tools are FIG (foreign interface generator) that takes C header files and a declarative script as input [RS06]. This allows the user to tailor the resulting interface to the application. The scripting language contains composable typemaps which describe the mapping between high-level and low-level types. The `ml-nliffigen` tool generates ML glue code from C declarations to enable the manipulation of C data structures from the ML application. Chasm is a toolkit wich provides seamless language interoperability between Fortran 95 and C++ [RSSM06]. Chasm uses the intermediate representation generated by a compiler front-end for each supported language as its source of interface information instead of an IDL. In this regard it is similar to our work as we also use the internal representation of the virtual machine for the generation of the wrapper code, but in our case dynamically, in the case of Chasm statically.

There is also other work of optimizing foreign function calls. Schärli and Achermann use partial evaluation of inter-language wrappers to optimize foreign function calls [SA01].

Persistent code caching has been used in a number of systems before, and is described in depth in the context of the Pin dynamic instrumentation system [RCCS07]. Fortunately, our code caching problem is much easier, and persistence is very easy to achieve in the absence of batching by using appropriate file names; batching makes persistent caching only a little harder.

7 Conclusion

In this work we present the implementation of a portable foreign function interface that uses dynamically-generated C wrapper files. We also present and evaluate a number of ways to reduce the startup overhead of this approach, and compare it to existing less-portable solutions like libffi and ffcall. Surprisingly, not only does our approach increase portability, it can also increase the performance: We saw speedups by up to a factor of 1.28 on application benchmarks.

References

[Bea96] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

[Bea98] David M. Beazley. Interfacing C/C++ and Python with SWIG. In *7th International Python Conference, SWIG Tutorial*, 1998.

[Bea03] David M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, July 2003.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ert07] M. Anton Ertl. Gforth's libcc C function call interface. In *23rd EuroForth Conference*, pages 7–11, 2007.

[ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference.

[FLMJ98] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/direct: a binary foreign language interface for haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 153–162, New York, NY, USA, 1998. ACM Press.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MMB+04] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanovic, and C. Weems. The DaCapo Project. http://ali-www.cs.umass.edu/DaCapo/, 2004.

[Ram03] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 6–14, New York, NY, USA, 2003. ACM Press.

[RCCS07] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Code Generation and Optimization (CGO '07)*, pages 74–88, 2007.

[RS06] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58, New York, NY, USA, 2006. ACM Press.

[RSSM06] Craig Edward Rasmussen, Matthew J. Sottile, Sameer Shende, and Allen D. Malony. Bridging the language gap in scientific computing: the Chasm approach. *Concurrency and Computation: Practice and Experience*, 18(2):151–162, February 2006.

[SA01] Nathanael Schärli and Franz Achermann. Partial evaluation of inter-language wrappers. In *Workshop on Composition Languages, WCL '01*, September 2001.

[SPEC99] SPEC Standard Performance Evaluation Corporation. SPECjvm98 Documentation. Release 1.03 Edition, 1999.

[Win97] H. Winroth. A scripting language interface to C++ libraries. In *Technology of Object-Oriented Languages and Systems, TOOLS (23)*, pages 247–259. IEEE Computer Society, 1997.