Porting of C library

Testing of generated compiler

Ludek Dolihal
Department of Information systems
Faculty of information technology, BUT
Brno, Czech Republic
idolihal@fit.vutbr.cz

Tomas Hruska
Department of Information systems
Faculty of information technology, BUT
Brno, Czech Republic
hruska@fit.vutbr.cz

***Abstract*— For testing the automatically generated C compiler for embedded systems on the simulator, it is necessary to have corresponding support in the simulator itself. Testing programs written in C very often use I/O operations. This can not be done without support of C library. Hence the simulator must provide an interface for calling the functions of the operation system it runs on. In this paper we provide a method that enables programs to run, which use functions from the standard C library. After the implementation of this approach we are able to use the function provided by the C library with limitations given by the hardware.**

Keywords - Porting of a library; C library; compiler testing; simulation.

I. INTRODUCTION

One of the goals in our research group is an automatic generation of C compilers for various architectures. Currently, we are working on Microprocessor without Interlocked Pipeline Stages (MIPS). To minimize the number of errors in the automatically generated compilers, it is necessary to put the generated compilers under test. Because the whole process of compiler generation is highly automatic and we do not have all the platforms, for which we develop, available for testing, we use simulators for compiler testing instead of the chips or development kits. If one wants to test the C compiler within any simulator, it is necessary to add the support for the C library functions into the simulator, which is used for testing.

The support of the library is crucial in our project. We need to use tests written in C for the compiler testing and the tests commonly use I/O functions, functions for memory management, etc. This paper presents the idea of fitting the simulator, where the testing is performed, with support of the C library and later on the implementation of this method.

The paper is organized in the following way; the second section provides the position of the testing in the Lissom project, then a short overview of related work is given, section four discusses the reasons for choosing the library. Sections five and six discuss the theoretical and practical side of adding the library support into the simulator. Section seven concludes the paper.

II. POSITION IN LISSOM PROJECT

In the Lissom project [1], we focus mainly on hardware software codesign. In order to deliver the best possible services we want to provide the C compiler for a given platform as C is one of the main development languages for embedded systems. The C compiler is automatically generated from the description file. Besides the C compiler there are a lot of tools that are also generated from the description file. The tools include mainly:

- simulators,
- assembler,
- disassembler,
- profiler,
- hardware description.

Simulators can be either cycle accurate or instruction accurate. The profiler was thoroughly described in this article [2].

The description file is written in ISAC [3] language. The ISAC language is an architecture description language (ADL). It falls into the category of mixed architecture description languages.

We would like to produce the whole integrated development environment (IDE) for hardware software codesign. This IDE should provide all the necessary tools for developers when designing embedded systems from the scratch. The simulators are part of the IDE and the C library is part of the simulators.

The tool for generating compilers is called backendgen and is also embedded in the IDE. The backendgen was programmed manually; it is not generated. The quality of a compiler is crucial for the quality of software that is compiled by the compiler. Hence it is very important to test the compiler that is generated by the backendgen. Through locating the errors in the compiler itself we can afterwards identify and fix problems in the generation tools and in the whole development process.

The primary role of the C library is to enlarge the range of constructions that can be used during the process of testing. Without all doubts it is important to test the basic constructions such as if statement loops, function calls, etc. On the other hand it is highly desirable to have a possibility of printing outputs or the exiting program with different exit values and this can not be done without C library support. Exit values are the basic notification of program evaluation and debugging dumps are also one of the core methods of debugging. Note that all the tests are designed for the given embedded system, and the tests are run on the simulator.

Secondary role of the library in the whole process of development is providing additional functions for writing programs. One of the most used groups of functions are functions for allocating memory, string comparison and parsing, input/output methods, etc.

As it is possible to generate several types of simulators in Lissom project, it will be necessary to add the library support into all types of simulators. It should not include any substantial changes to the process of generation.

III. Related work

Simulators in general are one of the most popular solutions as far as embedded system development is concerned. They are very often used for testing. We tried to pick up several examples that are connected to embedded systems development, and were published in the form of an article. The Unisim project is not aimed at embedded systems but provides an interesting idea.

In [4], a system very similar to the one that is developed within our project is suggested. It is called Upfast. The article describes system that generates different tools from a description file such as we do. The article mentions that C libraries were developed, but no closer information is given. It seems that in the simulator of the Unisim project the support for C language library has been right from the beginning. Unfortunately, this is not our case. Porting of the library is critical for us, because without the support it is very difficult to test and evaluate the results of any tests.

Another interesting system including a simulator is described in [5]. The project is called Rsim and is focused on the simulation of shared memory multiprocessors. The Rsim project works under Solaris. The Rsim simulator can not use standard system libraries. Unfortunately, it is not explained why. Instead the Rsim provides commonly used libraries and functions. The Rsim simulator was tested for support of a C library. All system calls in the Rsim are only emulated, no simulation is performed. In our system we will simulate the calls when necessary. The Rsim does not support dynamically linked libraries and our system also does not consider dynamic linking at the current state. Unfortunately, in the article [5] is not mentioned how the support for C library functions was added into the simulator.

The Unisim project [6] was developed as an open simulation environment which should deal with several crucial problems of today simulators. One of the problems is a lack of interoperability. This could be solved, according to the article, by a library of compatible modules and also by the ability to inter-operate with other simulators by wrapping them into modules. Though this may seem to be a little out of our concern, the idea of an interface within the simulator that allows adding any library, is quite interesting. In our case we will have the possibility to add or remove modules from the library in a simple way. But the idea from the Unisim project would make the import of any other library far easier than it is now.

The articles above are all related to simulations. The C programming language is not a new one and it is not possible to list all the articles that are in any way related to any library of C language. The simulator is either created in a way that it already contains the library or it has at least some interface which makes it easier to import the library in case it is wrapped in a module. Unfortunately, our simulator does not contain such an interface.

IV. Choosing the library

As we are focused mainly on embedded systems and we design the whole process of compiler development for them we dedicated quite a lot a time to choosing the correct library. It was clear right from the beginning that glibc is needlessly large and therefore not suitable for use in embedded systems. We need a library that satisfies the following criteria:

- minimalism,
- support for porting on different architectures,
- well-documented,
- new release at least once a year,
- compatibility with glibc,
- modularity.

All these conditions were satisfied by very few libraries. Amongst those we chose uClibc [7]. This library is largely minimalistic. It does not contain certain modules, because, according to the authors, it would be against the principle of minimalism. In certain areas it sacrifices better performance in favor of minimalism. For example, functions for I/O could be optimized for different platforms, but there is just one version for all platforms written in portable C that is optimized for space.

V. Theory of porting

The main reason for porting the library into a simulator is the fact that we need to add the support for C functions into the simulator itself. To be precise, we want to use the libc functions such as printf, malloc, free, etc. in the programs that will be used for testing of the compiler. And because we do not posses the development kits for all the platforms on which we run our tests we use simulators instead.

If one does not grant libc library support in the simulated environment, the number of constructions we can use and test is very limited.

Consider the following simple example written in C:

```
int main(int argc, char **argv)
{
  if(strcmp(“alpha”,”beta”)==0)
{ return 1;}
  else
{ return 0;}
```

}

Even this simple program can not be executed, because it uses function strcmp that is part of the C library. This program can not be compiled unless the inclusion of string.h and possibly some other header files is performed.

On the contrary the main aim of testing is to cover as wide an area as possible and also try as many different combinations of functions as we can. However, this goes against the idea of embedded solutions. And because we focus especially on embedded systems, we do not even try to cover all the functions provided by glibc, or in our case, uClibc. In fact we will use and hence test only functions that can run under the simulated environment and are useful for the programs that will be executed on the given platform. Moreover embedded systems are not designed for use of vast numbers of constructions that programming languages offer. Typically there is just one task, usually quite complicated, that is launched repeatedly. The functions we will use forms just a small part of uClibc. The functions that are not important to us can be easily removed via a configuration interface or manually. The following categories are examples of unimportant functions:

- threads, we assume that in simple programs for embedded systems one will not use threads,
- locales, all the locales were removed from the library,
- math, functions for computing sin, cos, etc.
- inet module, even though networking plays an important part in modern embedded systems the whole module was removed,
- files and operations with files, our application does not need an interface for working with files.

Now we come to the important parts of the library. Simply spoken all that really has to remain from the library are the sysdeps, this is the core of the whole system (how to allocate more memory, etc.), then important modules such as stdio (for outputs, inputs) and other modules we wish to preserve. In our case we wished to preserve the following parts of the uClibc library:

- stdio, this was the main reason for porting the library, to get in human readable form output from the simulator,
- a module for working with strings and memory, in our applications we would like to use functions such as memcpy, strcpy, strcat, etc.,
- memory functions, for example malloc, free, realloc,
- abort, exit,
- support for wchar, but without support of different encodings.

Some parts of the library could not be removed because of the dependencies. According to our estimations nearly 40 percent of the library was disabled or removed, measured by the size of the library.

There are several ways of building the library and also different methods of using it. There is a possibility of building a position independent code (PIC). Even though this is an interesting solution we decided against it. Instead of PIC we are going to compile the library into a single object and then link it to the program statically. The position of the library in the whole process of testing is shown in figure 1.

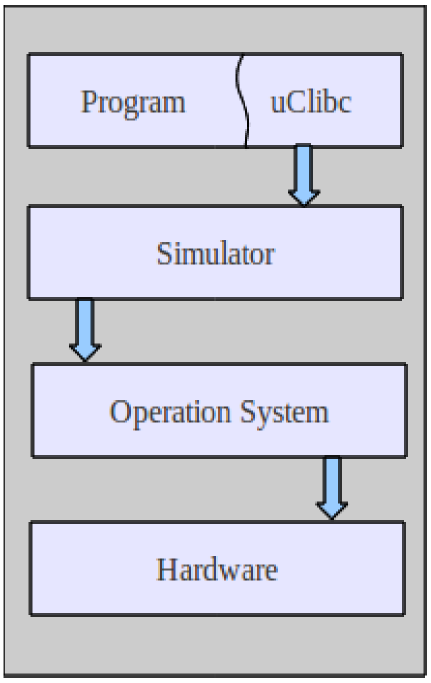

Figure 1. Position of the library in testing system.

Let’s return to the functions that remain in the library. They can be divided into two groups. The first group consists of functions that are completely serviced within the simulated environment. For example, function strcmp falls into this category. This function and its declaration remain unchanged within the simulator if it is written in portable C. These functions are not tied with kernel header files so there is no need to change them.

The second group of functions consists of functions that are translated to the call of system function. Function printf can be used as an example of this group of functions. The call of printf function can be divided into three phrases that are illustrated at the following picture.

In the beginning the call of printf function is translated into the call of the system function, with the highest probability it is going to be the call of function write. Write, being the POSIX function is offered by the operation system. But as we want to use the simulator on Unix platform as well as on Windows systems we have to get rid of these dependencies. To do so we will use the special instruction principle.

A. *Use of ported library of Unix and Windows systems*

Before we get to the principle of a special instruction method we should explain why we need to use this method.

The main reason why we should oust the dependencies on the kernel header files is the fact, that we must be able to use the library under Unix like, and also under Windows like operation systems.

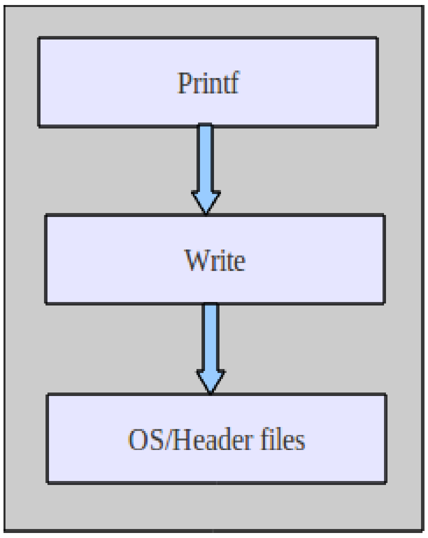

Figure 2. Scheme of calling the printf function

As long as we use the library under Unix systems everything should be all right. Though even on Unix systems there might be differences amongst the different versions of the header files. But once we use the Windows based system we can not use header file functions any more. It would almost certainly result in a crash of the system.

In our project we currently support several Unix distributions as well as Windows. Use of other operating systems is not considered.

B. *Special instruction principle*

The special instruction principle means, that we will use instruction with the operation code (opcode) that is not used within the instruction set of given microcontroler for the special purpose. We can do so, because we design the chip from the scratch. Usually the microcontroler has a given set of instructions. There is a defined bahaviour for each instruction. We describe the instruction set by our language ISAC. In ISAC, every instruction has an opcode and defined behaviour. So if there is any spare opcode we can model a new instruction with behaviour that suits our needs.

So far, all architectures that were modeled within the Lissom project had several free opcodes. It is typical that the instruction sets do not use all operation codes that are provided. But in case of no free opcode this method can not be used. The special instruction principle will be used for ousting the dependencies on kernel header files.

Functions provided by operation system are called by the system call (syscall) mechanism. The system calls can be quite easily detected. Each library should have defined the syscall mechanism in special source file. This syscall mechanism differs, as they usually are platform dependent. So i386 architecture will have different syscall mechanism than arm.

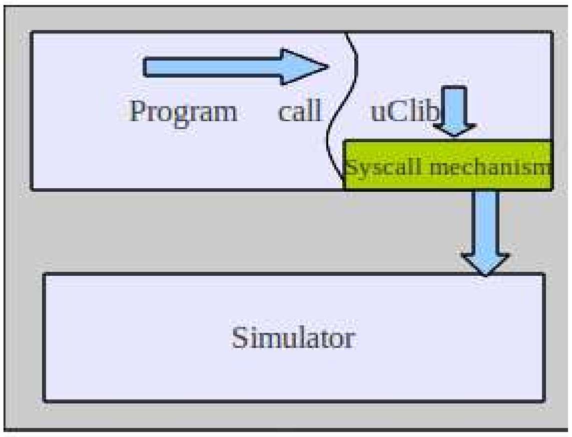

Figure 3. Scheme of calling the simulator via uClibc layer

We wish to preserve the mechanism. The syscalls will remain in the library, but with a different meaning. The file containing syscall will be changed in the following way: in the beginning the parameters of the syscall will be placed at the given addresses in the memory and we will also define where the syscall return value will be placed. Afterwards the call of the instruction, which was designed for this purpose, will be performed. It is also possible to put the parameters into registers, but some platforms have a limited number of registers, hence this method could cause problems.

C. *Simulators*

This brings us back to the simulators. As was mentioned before, all the simulators, where the testing is performed, are generated automatically. The generation is based on the instruction set description file, where our special instruction is modelled. In the beginning all the source files are generated by specialized tools. When the generation phase is finished the simulator is built by a Makefile from the generated files. It will be necessary to add into this process the following information:

- information about which instruction (opcode) calls the system function,
- the simulator will have to know the convention for storing parameters,
- the simulator will have to recognize which system function is going to be called,
- the simulator will have to perform the call of the correct system function.

The first three points will be solved within the model of an instruction set. The instruction with the opcode that is not used will be declared. The instruction behavior will be defined in the following way: it will locate the position in the memory where the parameters are stored and according to the value of one of the parameters it will call the corresponding system function. The simulator will have to recognize the system it runs under and call the correct function. For example, in Unix system, it will be function 'write', and in Windows system, WriteFile. This should be solved by the libc library of the given platform. The following figure demonstrates the call of special instruction.

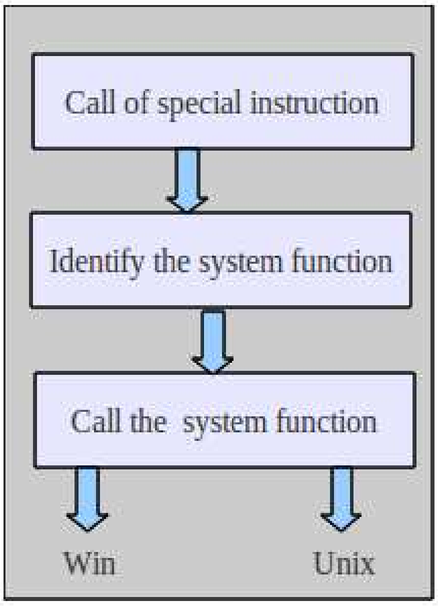

Figure 4. Calling sequence of specialized instruction

The parameters that were placed at the given position on the simulated memory can remain unchanged. They will later be passed to the specific system call.

One important issue is connected with the simulated memory. As we would like to correctly simulate the operations with memory such as malloc, realloc, etc. we need to tell the simulator how much memory it can simulate. This will be done most probably by the special file that will be passed to the linker. This file will contain symbols that will declare how much memory can be used. It is necessary to state how much memory can be allocated. The symbol that denotes the heap end will be used in the sbrk function.

VI. Process of porting

Before the whole process of porting begins we need to download the uClibc. There are two possibilities. It is possible to download only the library or there is a whole toolchain for development of embedded system for a given architecture, the so-called buildroot.

The main advantage of downloading the whole buildroot is that once it is built you get a whole set of development tools including various compilers, linkers, debugers, strip programs, etc. You also get the build of uClibc. These tools are quite useful in the beginning when you remove useless modules from the library, because they can be used for rebuilding the library.

One of the problems we faced is that we need to have the compiler for the architecture we are developing for. In other words, if we want to create a library for testing a C compiler on a given platform we need a compiler for the same platform that is already created. The compiler will be used for building the uClibc. Moreover the compiler must have exactly the same instruction set. In the future we would like to use the generated compiler for building the library. This requires a high quality of backendgen and generated backend.

Because we are going to use the library in the simulator and the simulator can handle only instructions of the specified instruction set, then the library must be translated to the instruction set that is recognized by the simulator. For building the simulator, we can use common gcc for Windows or Unix, because it runs under common system such as Windows or Unix.

This may be the first big problem in the whole process of porting. It is not hard to find a compiler for a given platform. Nowadays, there are specialized compilers for nearly all architectures used in embedded systems. The buildroot for uClibc contains more than a dozen different architectures such as MIPS, arm, mipsel, sparc, etc. There are even different versions of the micro-architectures in case of Microprocessor without Interlocked Pipeline Stages (MIPS), for example.

The problem is that, thanks to the aim of the whole Lissom project, we usually use specialized instruction sets or we use some generic instruction set and add certain specialized instructions. After this customization it is usually impossible to use a generic compiler for building the library.

We could use the compiler that we want to test for building the library but currently it is not stable enough for building large programs. The best solution of this problem is usually building a specialized toolchain including GNU binutils and GNU compiler collection [8]. As it was mentioned, once the generated backend is stable enough it will be used for building the library.

Several issues we faced during the process are closely related to the buildsystem of the library. The library contains a system of makefiles. This system is hierarchical and usually the makefiles from the upper levels are included. So, if for example we would like to compile any test examples that are included in the uClibc we switch to the given directory and call make. This will call all the makefiles from the above directory. This is very effective, because only the makefile in the root directory contains variables defining which compiler, assembler, linker will be used. On the other hand, it is very difficult to modify this system in case we want to build the different parts of the library using different tools.

Currently, we are using for development the set of our tools containing archiver, linker, asembler and compiler. The currently used compiler is called mips-elf-gcc. It is not generated automatically but was created specially for this purpose as our generated compiler is not stable enough for compiling the library. Our compiler has in the current version problems with floating point number, so it usually fails to compile them. Linker and archiver are not generated automatically for each platform but were developed in the Lissom project.

Our tools are not compatible with the tools that were originally used for building the library. Our tools do not support such a wide variety of parameters so some of them had to be erased from the configuration files and some were just changed to suit our needs. There are two main reasons for this. The first is that we simply do not need all the parameters. For example we always build files in elf format, so we do not need parameter to specify this. The second reason is that our files have different (usually text) format, that allows us to debug more effectively.

Currently, we use a set of scripts, which preprocess the flags. In the scripts we erase the flags we do not need and make necessary substitutions.

The buildsystem of the library starts by parsing the configuration file and accord to the content of the file are set different macros and variables. When doing manual changes to the buildsystem we have basically two possibilities:

- change the configuration file or,
- do the changes later in the Makefiles.

The first possibility is cleaner but the Makefiles often check if the option is present in the configuration file and ends with an error if the option is missing. Hence it is more convenient to make the necessary changes in the Makefiles. Thanks to the hierarchical structure it is in most cases sufficient to make the change in just one place.

In the theoretical part, we mentioned the need to link a special file containing information on how much memory can be used. The file will contain symbols defining the beginning and the end of memory space that can be used. It will have the following syntax:

```
#file defining memory boundaries
define start 256
define stop 768
```

Given that the numbers are in kB the simulator can simulate up to 512 kB of memory. Character # denotes comment.

For storing the parameters we have chosen the following approach: the first parameter says which system function is going to be called. In the uClibc it is a list of system functions for Unix systems. The rest of the parameters, that have numbers 2-7, are passed to the function call. The parameters remain unchanged. They are passed to the system function in exactly the same state which were saved in the memory before calling the special instruction. The special instruction itself has no parameters. When the instruction is called, all the parameters have to be stored in the memory at given addresses.

A. *Automation of the porting process*

For the first time, all the steps were performed manually. In the future we would like to automatize this process as much as possible. Without doubt we could remove the needless parts of the library automatically. The needless parts would be identified by the configuration file and also the special instruction principle could be highly automatic. If we have spare instruction we will choose it and compose it into the simulator. Unfortunately, there are steps that need to be performed manually, for example, we need to provide the runtime file for the simulators and the corresponding sections need to be specified in the ISAC file.

File with the runtime is also one of the files that is written by hand in the assembler. There are also other files written in assembly language and hence are platform dependent. In case of mips platform there were 8 files that contained assembly language. For example syscalls or memcpy functions are implemented in the assembler. In order to minimize number of files written by hand we decided to provide as many files written in portable C as possible. We managed to replace all but two files by C implementations. All that has to be provided is the runtime and syscall mechanism.

VII. Conclusion

In this paper, was sketched the idea of porting the library into the simulator. The motivation is quite clear: to be able to use the library functions in the tests that are run on the simulator of the given micro-controller. The special instruction principle was proposed which enables us to forward the call of system function. It also allows us to identify which system function is called. This principle is quite universal and can be used for the majority of platforms. After implementation of this method, we are able to run all the functions that are commonly used, such as I/O functions, memory management and string functions, etc. Moreover we can adjust the library according to our needs. Thanks to the modularity we can enable or disable any module. This may turn out to be an advantage, because the complete library has tens of megabytes and compilation and linking such a library can be time consuming.

Ackwnowledgements

This research was supported by doctoral grant GA CR 102/09/H045, by the grants of MPO Czech Republic FR-TI1/038, by the grant FIT-S-11-2 and by the research plan no. MSM0021630528.

References

[1] Lissom Project, doi :http://www.fit.vutbr.cz/research/groups/lissom, [online, accessed 19. 4. 2011] .

[2] Z. Prikryl, K. Masarik, T. Hruska, and A. Husar, “Generated cycle-accurate profiler for C language,” Proc. 13th EUROMICRO Conference on Digital System Design, DSD'2010, pp. 263—268, in press.

[3] ISAC language, doi:http://www.codasip.com/, [online, accessed 19. 4. 2011].

[4] S. Onder, and R. Gupta, "Automatic generation of microarchitecture simulators," Proc. 1998 International Conference on Computer Languages, May 1998, pp. 80-89, in press.

[5] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve, "Rsim: simulating shared-memory multiprocessors with ILP processors," Computer , vol.35, no.2, Feb. 2002, pp. 40-49, in press.

[6] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," Computer Architecture Letters , vol.6, no.2, Feb. 2007, pp. 45-48, in press.

[7] Uclibc, doi:http://uclibc.org/, [online, accessed 19. 4. 2011].

[8] GNU Operating System, doi:http://www.gnu.org/software/, [online, accessed 19. 4. 2011].