Testing of an automatically generated compiler

Review of retargetable testing system

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

Tomáš Hruška, Karel Masařík
Department of Information systems
Faculty of information technology, Brno University of Technology
Brno, Czech Republic
{hruska, masarik}@fit.vutbr.cz

***Abstract*— for testing automatically generated C compiler for embedded systems on simulator, it is necessary to have corresponding support in the simulator itself. Testing programs written in C very often use I/O operations. This functionality can not be achieved without support of the C library. Hence the simulator must provide the interface for calling the functions of the operation system it runs on. In this paper, we provide a method that enables running of programs, 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. Moreover we add the overview of the testing system, which is used in our project. The system allows testing hardware and also software part of the project.**

Keywords - Porting of a library, C library, compiler testing, simulation, hardware/software codesign, Codasip.

I. INTRODUCTION

This article is closely related to the paper [1] published at the ICCGI 2011. It will discuss the problematic of testing of the automatically generated compiler more closely, will focus on all major stages of compiler generation and on testing of the stages. As the main aim of the Lissom project [2] (commercialized under the registrated mark Codasip - www.codasip.com) is hardware software codesign we have to test not just the software part but also the hardware part.

One goal of 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 the 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. In order 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 the testing. The C programming language is still one of the most used languages for programming of embedded systems. Hence it is important to provide the reliable C compiler to the developers.

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. Second section provides the position of the testing in the Lissom project. After that we sketch the concept of retargetable testing system. Overview of the current stage of the testing is provided in section four. Then the short overview of related work is given, section six discusses the reasons for choosing the library. Sections seven and eight discuss theoretical and practical side of adding the library support into the simulator. Section nine describes the process of testing. Section ten presents the results obtained from commercial testsuite and finally section eleven concludes the paper.

II. RELATED WORK

As the core of the paper is dedicated to the testing of the compiler in the simulator we will focus mainly on related work in this area.

Simulators in general are one of the most popular solutions as far as embedded systems 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 a form of article. The Unisim project is not aimed at embedded systems but provides interesting idea.

Paper [6] presents a system that is very similar to the one that is developed within our project. 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 have 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 simulator is described in [7]. The project is called Rsim and is focused on 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 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 this article is not mentioned how the support for C library functions was added into the simulator.

Unisim project [8] 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 the 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 different ways of compiler testing of any language are listed in [13]. 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 interface.

III. Position in LISSOM project

In the Lissom project 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 the C language 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.

The simulators can be generated either from a cycle accurate or an instruction accurate model. The profiler was thoroughly described in [3].

The description file is written in ISAC [4] language. The ISAC language is an architecture description language (ADL). It falls into the category of mixed ADL.

We would like to produce the whole integrated development environment for hardware software co-design. This IDE should provide all the necessary tools for developers when designing embedded systems from the scratch. The simulator is part of the IDE and C library support is part of the simulators (in the IDE can be more that one simulator).

The tool for generating compilers is called *backendgen* and is also embedded in the IDE. The quality of a compiler is crucial for the quality of software that is compiled by compiler. Hence it is very important to test the compiler that is generated by the *backendgen*. Via locating errors in the compiler itself we can afterwards identify and fix problems in the generation tools and in the whole process of development.

The *backendgen* closely cooperates with the semantic extractor. The semantic extractor as the title suggests, extracts the semantics of the instructions specified in the ISAC file and after that the *backendgen* creates backend of the compiler that recognizes given instructions. Both these phases of the compiler generation will be discussed later on.

The primary role of the C library is to enlarge the range of constructions that can be used during the process of testing. Testing of basic constructions such as if-statement, loops or function calls is important. On the other hand it is highly desirable to have a possibility of printing outputs or exiting program with different exit values and this can not be done without a C library support. The 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. The tests are aimed mainly on robustness of the system.

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

As it is possible to generate several types of simulators in the 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.

IV. Concept of the retargetable testing system

Forget about the technical details for a while and let us have a closer look at the concept of the testing system. We should define the goals we would like to achieve with our testing system. The Lissom project should have a robust system of testing that is built modularly. As the system should support hardware as well as software testing it should be composed of two main modules.

The very first question that should be answered is what parts of the project we need to test. The main aim and focus of this article is on the testing of the compiler backend. But there

are also other parts of the project that should be tested. The hardware realization of the chip, that was mentioned above is one of them. Also important is testing of tools that are not directly connected to the compilation toolchain, for example disassembler. This leads us to dividing the software module into two separate modules.

The testing system should be multiplatform and highly modular and also highly configurable. The addition of the new platform that should undergo the tests should be trivial. The microprocessors that we are going to test can vary in many ways. We need to support all these features of the microprocessors.

The task, for which the embedded system is going to be developed varies widely. On the other hand the tools that will be used for the development will stay more or less the same in all circumstances. This leads us also to the idea of the core system and many modules that should be optionally connected into the process of testing via interfaces.

As it was mentioned in the section 2 we can have either cycle or instruction accurate model. For the full testing we should have both of them. Full testing here means testing hardware as well as software part. Unfortunately it is not always possible. The testing system must reflect this and be able to adjust the testing to the actual conditions.

As far as the software testing is concerned we should take into account the different levels of compiler optimalization as certain errors can be sensitive to this.

It is crucial to work with the most up to date tools so interface to any version system is a must. There should be also other interfaces, mainly the output ones. The system should be able to automatically inform a user about the result of the testing. There should be the email interface to send the result of testing to the person that performs it. We can also argue about interface to a bug tracking system such as Bugzilla or Trac. Though this interface would allow us to report the bugs automatically there is a risk of flood of false reports (the situation that one problem triggers others). Another issue is connected with the information that should be filled when the bug is created.

This problem could be solved by addition of a database between the testing system and the bug reporting tool. In the database we could keep records about the bugs that are currently reported and not yet fixed, hence we could avoid the redundancy of the bugs. Once the bug is fixed we could invalidate the database entry and if the same problem occurs again it could be reported again.

The notice about most up to date tools used for the testing leads to one module. The core module should responsible for creation of all possible tools but not for testing of any kind. It should just verify that all the source code is valid and that tools can be created. Between the phase of creation of the tools and the testing of the tools is clearly defined interface. These two parts can be run separately.

Right from the beginning we should take into account that all our tools can be used under both UNIX and Windows operation system. This is not a problem as far as the high programming languages are concerned (such as C or Java) for the programming of the devices. However the testing, which is the same in this case as running the testsuite should also be possible on both operation systems. And as the testsuite is created in Bash we must provide the basic support of the UNIX tools also under Windows. The solution here can be either MinGW or some other support such as Cygwin.

This brings us to the choice implementation language for the testing system. Unfortunately, the high level programming is not suitable for this kind of project. The testing involves an editing of various files, creating (make-ing the tools) control of return values and so on. Mainly for this reason, we chose the scripting in Bash as the best possibility. This brought us some difficulties as we will see later.

Our users will also use a different operation system and also different distributions of the UNIX systems. So from the beginning we must consider this. Not only different operation systems and also different releases must be taken into account as the there might be different versions of the GCC compilers for example. The only way, we can sufficiently handle this is virtualization of the machines where the testing system will run.

At least some of the components of the testing system should be usable separately. It would be without all doubts useful to run just testing without the prior build of the tools. The tools can be built via the graphical interface for example. Dually, we can encounter a situation when the build of tools is sufficient and no testing should be performed. Arguably, the likelihood of the first case is higher. It is also given by the fact that there are several ways of building development tools.

Hence the module for the testing itself should stand alone and should have the clearly defined interface. For the thorough testing we should have as many tests as possible. Unfortunately, this goes against the principle of embedded systems. The microcontrollers often have very reduced instruction set, so the chips are not capable of executing the tests. Therefore, we need a system of the test selection that will ensure that just the clearly defined subset of tests will be compiled and executed for the given platform.

Hand in hand with the selection of the tests goes their evaluation. The selection of the tests should be centralized as much as possible. On the contrary, the evaluation of the tests can not be centralized thanks to the different testsuites we use in our project. They have different formats of the output and also exit codes differ in the meaning.

Together with the results and evaluation goes an issue connected with the reporting of the errors. Once we encounter an error and we want to report it we should know who is responsible for the error (or which tool generated the error). This could be determined via testing the tools separately. In case of testing all the tools together, we can rely just on error messages and on temporary files that could be created. By the temporary files we mean the files that are output of one tool and input of the very next tool.

V. Overview of the testing system in LISSOM project

At this point, I would like to give an overview of the testing system in the Lissom project. It should give the reader more precise information about the whole system and how the library fits into the whole. Our testing system is written in the Bash language. It consists of set of scripts. The testing system was originally developed for the UNIX systems. Should it also work under the Windows, it is necessary to support in the form of the MinGW. This approach brings on problems such as different paths on various systems or different settings of environment variables that have to be dealt with.

The testing system or testsuite as it is called in our project performs four basic tasks:

- testing of the tools for the development,
- testing of the backend of the C compiler,
- hardware testing,
- creation of the releases and packages of the models.

Now let us have a closer look at the parts of the project one by one.

A. Tools for the development

As far as the testing of the tools for the development is concerned it consists of several phases, which should be performed in given order.

As we always need to work with the most up to date tools the first thing that must be performed is the download of all necessary source code from repository.

The first phase is a build of all the tools. Even though the advanced IDE are used during the development very often happened that the source code can not be compiled.

Once the tools are created, the testing phase begins. We perform the testing of each tool and also testing of the toolchain to make sure that the cooperation is guaranteed.

Some of the tools such as assembler or simulator are platform dependent. So we have to keep in the repository the source codes for the testing for each platform. For the architecture independent tools this costs can be saved. The same problem occurs for the reference output. Certain tools can also have different levels of optimization and/or generation of information for profiling. Thanks to this fact the number of reference results grows rapidly. Currently, we are working on the new version of the testing system, and one of the tasks is to lower the number of reference outputs. Another weakness is that we do need the reference output. It is usually gained manually.

As mentioned earlier, we have different kinds of simulators. We perform testing on all kinds of simulators with all possible levels of generation of profiling information. The amount of generation of profiling information can be specified during the simulator generation. This is in contrast with testing of the compiler backend, where we use just one simulator and generate minimal amount of profiling information.

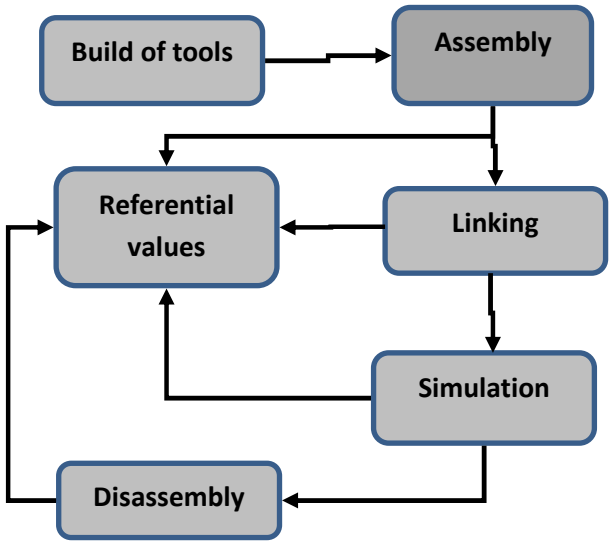

Figure 1. The scheme of testing of the development tools

We also perform tests that ensure the integrity of the whole system and a compatibility of the tools. In other words, we must ensure that if we add some new features into one of the tools the rest of them will be able to cope with these changes.

Typically, we bring some testing input written in an assembly language to the assembler and go through all the phases. In the end we should gain the executable file and be able to run it in the simulators with the correct return value. We also try to disassemble the executable. The code we receive should have the same functionality as the source one.

It may seem that both mentioned approaches are the same. However, the crucial difference is that while in the first case the tested component can go through the testing process without errors, there can be some issues connected with the file formats and interfaces between the tools. The first way of testing is on the other hand used for experiments with new features of particular components that are not supported by the whole toolchain yet. Figure one shows the process of testing of development tools. We have a simple program in C usually. This program goes through the whole toolchain. It is assembled, linked, simulated and in the end disassembled. After each stage we compare the result and referential value.

The hardware testing is also performed in this module. However we automatically perform just the tests of the syntactic correctness. No workbenches are executed.

B. C compiler backend

As far as testing of the compiler is concerned we first need to create the compiler and compiler driver. After that we can start testing. Here we will describe the process of the compiler generation and creation of compiler driver. The testing process itself will be thoroughly described later.

The LLVM project [10] is used by our research group as a base we build on. LLVM stands for low level virtual machine. It is a project focused on creation of modular compiler that provides aggressive optimalization. In fact, the frontend and the middlend part of the compiler are used without massive changes. The part that is crucial from our point of view is the compiler backend.

The backend part is responsible for printing the assembler. This part is generated automatically by *backendgen*. As we use certain parts of the LLVM with no or small modifications we added the Lissom target into the LLVM project. This way we build programs that are later used for compilation of source code. Namely we create Clang this way. Clang is a frontend of the compiler provided by the LLVM project.

Given that we have built the LLVM project, we can start with the creation of compiler backend. This phase begins with the semantic extraction. As mentioned before the input of the whole process is file written in ISAC language. From the file that represents instruction accurate model of a microcontroller we extract semantics. After this phase, we get file that captures meaning of the instructions. More precise information about the semantic extraction phase can be found in the article [5].

The file with the extracted semantics is one of the inputs of the backend generator. The *backendgen* generates source files mainly in the C language that are later on compiled by ordinary C compiler (ie. gcc). As it is generated from the model it is clear that compiler backend is platform dependent. The semantic extractor and backend generator very closely cooperate. After the successful generation of the compiler backend we can create the compiler driver. The other tools that are required for the translation process are generated from the model before the backend is created. The brief overview of the backend generation can be found in here [5].

While the generation of all the tools is a must for the test compilation, it is not compulsory to build the compiler driver, but it simplifies the translation process considerably. The gcc compiler is in fact also compiler driver. We use compiler driver provided by the LLVM project. It is called *llvmc*. The tools that are used, parameters that are accepted by the tools and also the order of execution are described by the given syntax. The *llvmc* description has three parts. The first part is the description of the tools that are going to be used. In the second part, one must provide the languages (and its suffixes) that are the input and the output of each tool. Finally, we specify the relations between the tools. We can think of it as a graph. The tools became the nodes and we can think of the relations as of edges. The input and output languages are properties of the nodes.

We also use compiler-rt project of the LLVM. The compiler-rt project provides implementations of the low-level code generator support routines. This routines and calls are generated when a target does not have a short sequence of native instructions to implement a core IR operation. In fact, when the compiler does not know how to achieve certain behavior with the given instruction set it has a look at the compiler-rt library whether there is a call that could be used.

The main part of the library is composed around the floating point arithmetic. The functions have single float precision (which is denoted by the sf in the name of the function) and also double float precision (denoted by the df in the name of the function). As our processors do not usually have its own instructions for floating point arithmetic we very often use this library to provide the floating point emulation.

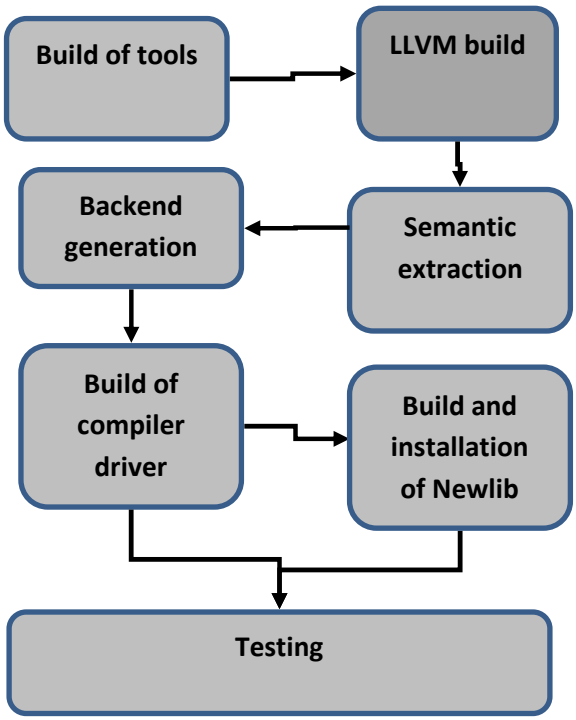

Figure 2. Scheme of testing of the compiler backend

The second figure shows in what order are the phases of backend testing executed. The libraries are not integral part of testing. It is possible to run the testing system without them.

The compiler-rt is for us just another library. We link it statically during the test compilation together with *newlib* for example. One of the issues is that this library is aimed at 32-bit systems. We would like to use it in simulators that simulate behavior of 16-bit processors. This has not been tested yet.

C. *Packaging and releases*

This module is a part of the testing system from the beginning. It was originally created for the building of the packages. As we currently support rpm distributions as well as deb distributions and also the Windows, the packaging system must reflect that.

The packaging system automatically creates the packages for the currently supported platforms. The package includes all the tools that are needed for the development on a given platform. Currently we support Ubuntu and Debian releases, Fedora, CentOS, OpenSUSE and Windows 7. For the majority of the UNIX distributions, we maintain the current release and previous one. All this systems run as virtual servers. The

created packages are automatically uploaded at our web pages, where can be downloaded by our co-developers and users.

Later also the packaging of the models and the model documentation were added. These are also available from the web pages. We have also started with nightly builds to ensure, that the committed changes do not affect the build in a negative way. Yet another advantage of nightly builds is that if any package is needed with the changes that were made within the last 24 hours the package was already created overnight and we do not have to wait for it to build. We can be uploaded it at the server where it is available to the customers.

D. Stability of the system

For a long time, we had problems with the stability of the whole testing system. As it is composed purely of the Bash scripts, we very often faced the problem, that one part of the testing system broke down according to an error but the build went on and the error was lost deep in the logs. This was typical situation during the night build, when the system is unobserved. In the morning we realized, that the packages were not created and started to look for the reasons. As our system creates a lot of logging information it was not always easy to identify the reason.

To solve this problem we decided to create simple wrapper and wrap all the commands except the calls of our procedures and functions. The wrapper performs the command that is given to it as a parameter and controls various variables. Clearly one of the most important is the return value of the command. If the return value is out of range we simply call system exit and the whole process stops with the clearly specified error message.

Even more important is the fact that we know the exact place where the error occurred. Unfortunately, this is not true in case we apply parallel build. But the wrapper is applied on the Bash commands so we at least know the command where the error occurred hence we can narrow the area and focus on the command more precisely. After the application of the wrapper the stability of the whole system improved.

VI. 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 of 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 library that satisfies 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 few libraries. Amongst those we chose Newlib [9]. This library is largely minimalistic. It does not contain certain modules, because, according to the authors, it would be against the 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.

As far as the new releases are concerned, it can be said that the library is alive. New version is released at least once a year. This is very important because we need to keep pace with the up to date versions of glibc. There are other minimalistic libraries compatible with glibc, but quite a lot of them are not maintained sufficiently.

Another reason for choosing the newlib is the documentation that is provided with the library. Whole process of porting the library to different platform is well-documented and thanks to the wide use of the library it is not difficult to find help.

The most important reason for choosing the newlib is the fact, that it has already been ported to several platforms. One document is dedicated to the process of porting and even though we do not port the library to new architecture it can provide us with very useful information. During the process of porting we will perform steps that are similar to porting the library to any new architecture.

Unfortunately this library is dependent on kernel header files. But during the porting we will get rid of these dependencies. We will need to use this library under UNIX systems as well as under Windows.

VII. THEORY OF PORTING

The main reason for porting the library into 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 possess the development kits for all the platforms 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 included.

On the contrary the main aim of testing is to cover as wide 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 newlib. 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 number of constructions that programming languages offer. Typically there is just one task, usually quite complicated, that is launched repeatedly. As we will see the functions that we will use form just small part of newlib. The functions that are not important to us can be easily removed via configuration interface or it is possible to remove them manually. 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 important part in modern embedded systems whole module was removed,
- files and operations with files, our application do not need 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 following parts of the newlib library:

- stdio, this was the main reason for porting the library, to get in human readable form output from the simulator,
- 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. Even though this is an interesting solution we decided against it. Instead of PIC (position independent code) we are going to compile the library into single object and then link it to the program statically. The position of library in the whole process of testing is shown in the figure 3. The library is linked to the program and after that the program is loaded into the simulator.

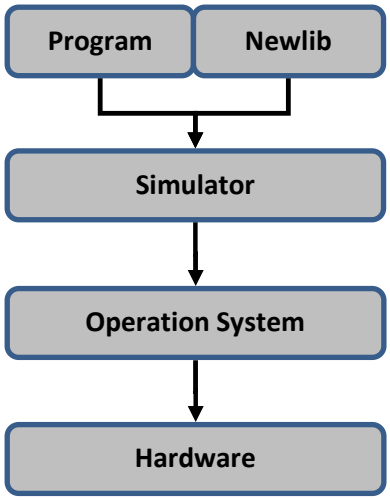

Figure 3. Scheme of calling printf function

Now return to the functions that remain in the library. They can be divided into two groups. 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 remains 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 on the call of 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 remove 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 special instruction method we should explain why we need to use this method. The main reason why we should remove the dependencies on the kernel header files is the fact, that we must be able to use the library

under UNIX systems and also under Windows like operation systems.

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 opcode that is not used within the instruction set for the special purpose. 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 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.

Syscalls together with other code that is platform dependent are kept in a specific folder. When the library is compiled, the platform dependent code is kept in a special archive and is separated from the platform independent code. Figure 4 shows this situation. We must link two different archives to the program we wish to execute. The C library and the archive containing syscalls and other platform dependent code such as runtime etc.

We wish to preserve the mechanism. The syscalls will remain in the library, but with 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 chosen instruction will be performed. It is also possible to put the parameters into registers, but some platforms have limited number of registers, hence this method could cause problems.

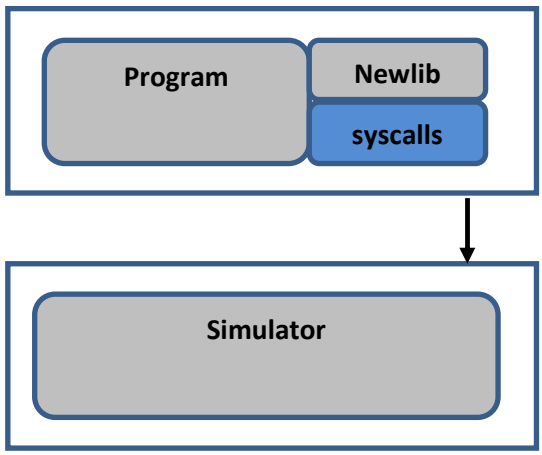

Figure 4. Scheme of calling the simulator via newlib layer

The syscall mechanism is in fact a wrapper of the system call. The call will be passed to the simulator that will do the call and return the result.

C. *Simulators*

As was mentioned before, all the simulators are generated automatically. In the beginning all the source files are generated by specialized tools. When the generation phase is finished the simulator is build by a Makefile. It will be necessary to add into this process 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.

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: according to the parameters it will call the given system function. The simulator will have to recognize the system it runs under and call the correct function. For example on UNIX system it will be function write and in Windows WriteFile. This should be solved by the libc library of the given platform. The following figure demonstrates the call of special instruction.

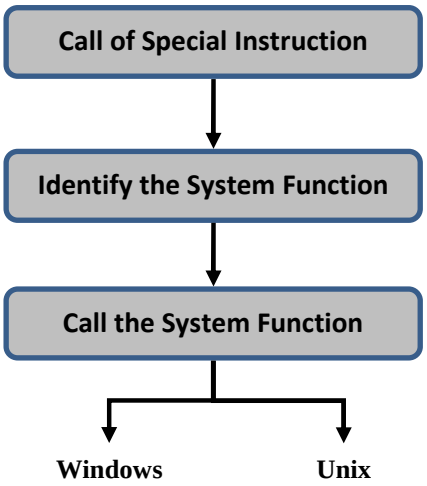

Figure 5. Calling sequence of specialized instruction

When the special instruction is called, we need to identify, which system function we need to execute. This information must be passed out of the simulator.

The parameters that were placed at the given position at the simulated memory can remain unchanged. They will be later 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 many memory it can simulate. This will be done 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.

We also considered completely different attitude to this problem. Instead of monitoring calls of system function we could monitor memory accesses. But it would slow down whole process of simulation.

VIII. PROCESS OF PORTING

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

The main advantage of downloading the whole buildroot is that once it is built you get whole set of development tools including various compilers, linkers, debugers, strip programs etc. You also get the build of newlib. These tools are quite useful in the beginning when you remove unwanted 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 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 newlib. 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 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.

This may be the first big problem in the whole process of porting. It is not hard to find a compiler for given platform. Nowadays there are specialized compilers for nearly all architectures used in embedded systems. The buildroot for newlib contains more than dozen of different architectures such as MIPS, arm, mipsel, sparc etc. There are even different versions of the mircoarchitectures in case of MIPS for example.

Problem is that thanks to the aim of the whole Lissom project, there 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 generic compiler for building the library.

We could use for building the library the compiler that we want to test 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. As was mentioned once the generated backend is stable enough it will be used for building the library.

Several issues we faced during the process were 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 newlib 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 the 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 especially for this purpose as our generated compiler is not yet stable enough. Linker and archiver are not generated automatically but were developed in Lissom project.

Our tools are not compatible with the tools that were originally used for building the library. Our tools do not support so 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.

Currently we use set of scripts, which preprocess the flags. In the scripts we erase the flags we do not need and do 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 error in case the option is missing. Hence it is more convenient to do the necessary changes in the Makefiles. Thanks to the hierarchical structure it is in most cases sufficient to do the change in just one place.

We also use different formats of the output files. Output of our assembler is an object file .obj that is not compatible with .o that is the usual output of gcc compiler. Currently we use mips-elf-gcc just for compilation from C to assembler. After this phase we use automatically generated assembler to compile the files from assembly language to object files that are later used by the archiver.

In the theoretical part we mentioned the need to link special file containing information 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.

As far as the convention for storing parameters is concerned, we have chosen following approach: first parameter says, which system function is going to be called. In the newlib it is a list of system functions for UNIX systems. The rest of the parameters (2-7) are passed to the function call. The parameters remain unchanged. They are passed to the system function in the exactly same state, in 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

As 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 doubts 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 for the simulators and the corresponding sections needs to be specified in the ISAC file.

Runtime is also one of the files that are written by hand in 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 ale implemented in assembler. In order to minimize number of files written by hand we decided to provide as much files written in portable C as possible. We managed to replace all but two files by C implementations. All that have to be provided is the runtime and syscall mechanism.

IX. PROCESS OF TESTING

Now when we have thoroughly gone through the library porting, we can have a look at the test selection issues.

A. Test selection phase

As we have a large amount of tests from the different sources (gcc-testsuite, llvm-testsuite, etc.), we need a universal approach that will define, which tests are suitable for the compilation and execution on a given platform.

We have created a system of files that restricts the number of the tests that can be compiled on a given platform according to the libraries that are available. The libraries are just one of the test selection criteria; also other characteristics are taken into account for example the size of the registers or the size of stack.

The naming convention for these files is very simple. The file bears same name as the test does but have suffix .x instead of .c. The system is hierarchical. We can have the hierarchy because we support a nesting of the directories and we keep .x files not just for the tests, but also for the directories. In case of the directory the .x file has the same name as the directory with the .x suffix.

These files have minimal functionality. We try to keep their size as small as possible. Their typical functionality is that according to some state of the flags the test is excluded from testing, because implicitly all the directories and all the tests are selected for the testing. So, if we want to exclude the tests or whole directories from testing we have to indicate this.

As the size of the files is kept minimal the functionality and flag settings must be done elsewhere. This is performed centrally in the main testing module. The functions that check the current state of the flags and control what libraries are accessible for linking to the given platform are declared here. The centralization in this case has purely practical base. The typical usage of the .x files is that we disable testing of the whole directories according to the libraries that are accessible. The .x files can also bear other functionality. We can for example set different variables. We can specify flags that should be added to the compilation or add some files to the linker as in the following example.

```
if [ "$C_LIB" == "0" ]; then
        FILE_DEPS+=crt0.o
fi
```

On the level of files we most often use the .x files for the filtering the test that depend on the compiler-rt library for a given platform. As usually only few tests of any directory depend on the compiler-rt and the dependence does not have to be same for all platforms, the best solution is to condition the test execution by the platform and the compiler-rt presence. This is demonstrated in the following example.

```
is_arch "mips_basic" $1
if [ "$?" == "0" ]; then
  if [ "$RUNTIME_LIB" == "0" ]; then
    RUN_TEST=0
  fi
fi
```

The presence of certain libraries can be also criteria for testing because some tests have library dependencies. The biggest advantage of this approach and also the main reason for introduction of this system is its universality. We employ the tests from the LLVM testsuite, the gcc testsuite, the Mibench [11] set of tests and we also have tests that were created within our project. The system of the .x files can be used for all these sources as long as we use just the tests without the testing infrastructure that is provided in several cases.

The only set of tests that we use together with the infrastructure that is provided together with the tests is the Perennial testsuite [12].

B. Test compilation and execution

The compilation of the tests is performed in the central module. As we have the system of the .x files we enter only those directories that are suitable for the testing on the given platform. So before entering the directory with tests we check the .x file for a given source and consult the restrictions that are defined by the .x file and set all the variables denoted by the file.

If the directory is feasible for testing we cycle through the tests in an order denoted by the test list. The .x file is always checked first, and if nothing blocks the test it is compiled. The presence of the .x files is not compulsory. As mentioned earlier the default setting is to go through all directories and execute all tests. But if the file is present it will be checked.

If there are any problems during the test compilation they are logged. We keep the list of the tests that were not compiled successfully together with the output of the compiler. The logs are kept for every platform that is tested to avoid an overwriting. It is also possible to create unique log not just for each platform but for every run of the testing system. These logs could be in the future stored in the database to keep precise testing history. The tests are compiled and executed several times with different levels of compiler optimalization. Currently we support levels from 0 up to 4.

C. Logging information and test evaluation

The test evaluation is kept decentralized. As we deploy tests from different sources we need to keep the test evaluation together with the tests. For some tests we evaluate on the basis of exit code, but there are the tests that produce for example text output and we have to compare the output with the referential values (this is where the library comes to use).

The decentralization in this case means that we keep for every directory a shell script that takes care of test execution and evaluation.

As in case of test compilation we keep detailed logging information. We keep the output of the simulator and after the test evaluation we list it into the list of passed tests or failed tests according to the result of evaluation. The logs are created for every tested platform and can bear time reference.

X. RESULTS OF PERENNIAL TESTSUITE

For having a comparison with commercial compilers we tested our automatically generated compiler with commercial Perennial testsuite. The results described here were gained from the generated MIPS compiler.

The testing was performed on our complete toolchain. The tests were compiled by our generated compiler and afterwards executed the tests on our simulator that was also automatically generated.

We have only part of the Perennial testsuite. We used only the tests that examine the core of the compiler. We excluded some of the tests that can not be compiled due to the header files dependencies we do not support. The tests in the testsuite are divided into groups according to the chapter of the standard that is tested. We use tests for the clauses 5 and 6. We have mainly tests for the standard C90 and several tests for C99 standard. Currently wa have no tests for C11 standard.

The final number of tests that we execute is 796. From 796 tests are 794 tests compiled and executed correctly. Only two tests fail either during the compilation or return incorrect value. The results are summarized in the following table.

Table 1: Results of the Perennial testsuite

Compiler	All tests	Pass tests	Fail tests	Not compiled	Not executed
Lissom	796	794	2	2	0
Gcc	796	796	0	0	0

As the table shows, just 2 tests do not succeed. After closer look we realised that this two tests use trigraphs, that are not supported in the llvm frontend. This tests can not be compiled by the current version of the llvm. The tests were compiler with O2 optimalization.

The table also provides comparison with gcc compiler for i386 platform. The gcc compiler in version 4.6.3. compiles all the tests and the programs are executed correctly. We were also interested in how much time does the program spend by syscall execution. We compiled for our platform a program that accomplished MPEG decoding. The input and output

streams of the program were redirected into the files. The profiling of the MPEG decoder showed, that execution of syscalls took less than 2% of time.

XI. CONCLUSION

In this paper, we gave the overview of the testing system in our project and sketched the idea of adding the support for the C 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 microcontroller. 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 to be an advantage, because the complete library occupies tens of megabytes and the compilation and linking such a library can be time consuming.

We also tested our generated compilers with the commercial Perennial testsuite. We had only chosen a subset of tests that should validate the core of the compiler. The compiler was tested against the C90 and C99 standard with good results when we take into account the fact, that the compiler is generated automatically. The fact that we can easily compose new testing systems into our own together with the results we gained is encouraging.

ACKWNOWLEDGEMENTS

This research was supported by doctoral grant GA CR 102/09/H042, by the grants of MPO Czech Republic FR-TI1/038, by the grant FIT-S-11-2 and by the research plans MSMT no. MSM0021630528. This work was also supported by the IT4Innovations Centre of Excellence Project CZ.1.05/1.1.00/02.0070 and by the Artemis EU Project SMECY.

REFERENCES

[1] L Dolihal and T. Hruska,, “Porting of C library, Testing of generated compiler”, In proceedings of ICCGI 2011, Jun. 2011, pp.125-130,

[2] Lissom Project. http://www.fit.vutbr.cz/research/groups/lissom [online, accessed: 18.6.2012]

[3] Z. Přikryl, K. Masařík, T. Hruška, and A. Husár, “Generated cycle-accurate profiler for C language”, 13th EUROMICRO Conference on Digital System Design, DSD'2010, Lille, France, pp. 263—268.

[4] K. Masarik, T. Hruska, and D. Kolar, “Language and development environment for microprocessor design of embedded systems”, In proceedings of IFAC workshop of programmable devices and embedded systems PDeS 2006, pp. 120-125, Faculty of electrical engineering and communication BUT, 2006

[5] A. Husar, M. Trmac, J. Hranac, T. Hruska, and K. Masarik, “Automatic C Compiler Generation from Architecture Description Language ISAC”, Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10) -- Selected Papers, pp. 47-53.

[6] S. Onder and R. Gupta,, "Automatic generation of microarchitecture simulators," Computer Languages, 1998. Proceedings. 1998 International Conference on , vol., no., pp.80-89, 14-16 May 1998

[7] 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, pp.40-49, Feb 2002,

[8] 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, pp.45-48, Feb. 2007

[9] newlib. http://sourceware.org/newlib/ [online, accessed: 18.6.2012]

[10] C. Lattner and S.V. Adve, ” LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004

[11] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, “MiBench: A free, commercially representative embedded benchmark suite”, Workload Characterization, Dec. 2001, pp.3-14, doi:10.1109/WWC.2001.990739

[12] Perennial testsuite, http://www.peren.com/ [online, accessed: 18.6.2012]

[13] A.S. Kossatchev and M.A. Posypkin, “Survey of compiler testing methods”, Programming and Computer Software, Jan. 2005, pp.10-19, doi: 10.1007/s11086-005-0008-6