INFORMATION AND COMMUNICATION TECHNOLOGIES (ICT) PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D2.2 Prototype implementation

Version 1.0

Main Authors:
Nicolas Ayache, Roberto M. Amadio, Yann Régis-Gianas

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Abstract The CerCo project aims at building a certified compiler for the C language that can lift in a provably correct way information on the execution cost of the object code to cost annotations on the source code. This report provides an overview of the port from the Mips to the 8051 processor of the untrusted prototype of the CerCo compiler described in deliverable D2.1 [2]. As such it is an introduction to the actual deliverable D2.2 which consists of an untrusted, cost-annotating compiler from the C language to the 8051 assembly language written in the ocaml programming language.

Contents

1 Introduction

The goal of the CerCo[1] project is to build a certified compiler for the C language that lifts in a provably correct way information on the execution cost of the object code to cost annotations on the source code. As a first step, an untrusted prototype of the compiler that targets the Mips processor has been developed in ocaml. The general structure of the compiler follows closely that of the CompCert compiler[2]. The Mips processor was selected for its conceptual simplicity, the quality of its documentation, and the reliability of the emulator. Indeed, the Mips processor is widely adopted as a pedagogical tool in computer architecture courses.

We assume the reader is familiar with deliverable 2.1 [2] which describes the *compiler* and the *labelling approach* used to compute the cost annotations. By default, the intermediate languages, their translations, and the related annotation process are those described in deliverable D2.1.

Experiments with the Mips processor suggest that the labelling approach can handle a realistic C compiler with a moderate level of optimization (comparable to level 1 optimization of gcc). Following these preliminary and encouraging experiments, the decision has been taken to target a processor with a more industrial flavor and which is widely used in embedded software. The choice has fallen on the 8051 processor which is a well-established 8 bits processor. The decisive fact that led to the selection of this processor is that manufacturers can provide *precise timing informations* on the execution of its instructions. This is due to the fact that the architecture of the processor is rather primitive. For instance, there is no notion of pipeline (like in Mips) or cache memory. We refer the reader to [5] for an adequate tutorial on the 8051 processor.

An interesting and alternative choice which has not been pursued so far for lack of manpower would be to select a more trendy 32/64 bits processor used in embedded systems such as one of the Arm family. For such processors, the worst-case cost may deviate significantly form the real cost. For this reason, state of the art products such as those developed by AbsInt[3] build an abstract model of the architecture to predict costs which are as close as possible to the reality. In principle, it seems possible to connect the CerCo compiler to such tools and therefore to lift the information provided by the AbsInt tool on the execution cost of sequences of binary instructions to cost annotations on the source C code.

In this report we will focus on the differences between the Mips and the 8051 processors and on the way they affect the compilation chain. In particular, section 2 compares the Mips and 8051 processors, section 3 describes the implementation work, section 4 presents a list of current restrictions and some plans for future work, and section 5 provides our tentative conclusions at this stage of the project.

2 Comparison of the Mips and 8051 processors

This section presents the differences between the Mips and the 8051 processors that have an impact on the structure of the CerCo compiler. From a compilation point of view, the most problematic difference is the size of the memory cell. While Mips is a 32 bits processor where addresses are coded on 32 bits (and thus fit exactly in one memory cell), the 8051 is a 8 bits processor where addresses are coded on 16 bits. This raises two main issues:

[1]`http://cerco.cs.unibo.it/`

[2]`http://compcert.inria.fr/`

[3]AbsInt Angewandte Informatik, `http://www.absint.com/`.

- The input language, namely Clight, allows to manipulate integers of size 8, 16, or 32 bits. This raises no particular problem when targeting Mips because any of those fit in a single memory cell, but this is not the case in the 8051.
- The second language of the compilation chain, namely Cminor, does not differentiate between integers and addresses thus making the implicit assumption that they are both coded with the same number of bits, which is true in Mips but not in the 8051.

We address these two issues below along with two others concerning instruction selection and calling conventions.

2.1 16 and 32 bits integers

Since the size of the 8051's memory cells are 8 bits, we need to use two memory cells to represent 16 bits integers, and four memory cells to represent 32 bits integers. In the end operations on 16 and 32 bits integers are simulated by sequences of operations on 8 bits integers and this simulation may introduce branches and loops in the binary code that are not present in the source code. This raises the problem of how to account in the source code for the execution cost of such operations. To address this problem, we follow the approach described in deliverable D2.1 [2] for operations such as logical conjunction which introduce implicitly a branching in the object code. The idea is to pre-process the source Clight code so that the branching is made explicit and then to label it. Concretely, we have implemented a type preserving transformation over the Clight abstract syntax which replaces the 16 and 32 bits integers with C structures of two and four fields of type `char` (8 bits), respectively (see section 3 for some details). At the same time, the 'primitive' 16 and 32 bits C operations are replaced by a library of C functions operating over structures.

2.2 Pointers

In the Mips processor, addresses have the size of a memory cell. Thus, when targeting Mips, the Cminor intermediate language (adapted from CompCert) does not need to make a difference between integer variables and pointer variables. When targeting the 8051, we need to restore this difference because addresses are two words long. In practice, we move from an 'untyped' Cminor language to a 'typed' version where we distinguish the type of memory cells (8 bits) and the type of addresses (16 bits) (again, this is detailed in section 3).

2.3 Instruction set

An obvious difference between Mips and the 8051 is the instruction set. Mips instructions are almost in bijective correspondence with the basic C operations. On the other hand, 8051 instructions even when operating on 8 bits data are at a lower level. For instance, in the arithmetic operations one operand must be put in a special register called 'accumulator' which is also used to store the result. This difference complicates considerably the instruction selection process which corresponds to the transformation from RTLAbs to RTL. This transformation has been completely revised.

2.4 Stack and calling conventions

The 8051 has an internal stack that is used for storing the return address of a function (unlike Mips, 8051 has no return address register). We could use this internal stack for all stack operations (saving a value, storing parameters, etc), but its small size, at most 80 bytes, is not convenient as long as we consider general recursive C programs. There seems to be no standard conventions for the implementation of the stack in external memory (every manufacturer has its own conventions). Given this state of affairs, we decided to adapt Mips stack conventions to the 8051. Then we emulate the stack in the external RAM, beginning at the highest address and growing towards lower addresses. The emulated stack pointer is stored in two distinguished registers. When a function is called, we simply move the return address from the internal stack to our emulated stack. In this way, the depth of the recursive calls we can handle only depends on the size of the external memory.

3 Implementation

In this section we provide more details on the port of the CerCo compiler from Mips to 8051.

3.1 Clight

Clight without floating point remains the source language for which the CerCo compiler produces cost annotations. Support for floating point is not planned as embedded software running on the 8051 does not rely on it. General C programs can be compiled to Clight with the CIL tool.

The abstract syntax of Clight did not suffer any changes, neither did its semantics, that is still based on 8, 16 and 32 bits integers. However, before translating to the next language in the compilation chain, we perform a transformation converting every 16 (respectively 32) bits integers to a structure composed of two (resp. four) 8 bits integer fields. These fields represent the bits of the integer in the order of their significance. Every Clight operation on 16 and 32 bits integers is replaced by a call to a function emulating the operations on the structures. This transformation, called Clight*32 to* Clight*8*, is defined in the module `Clight32ToClight8` in the `src/clight` directory. Below is the list of the transformations it performs.

- The types of 16 bits and 32 bits integers (`short` and `long`) are replaced by structures of respectively two and four fields of 8 bits integers (named `_int16` and `_int32`).
- Every 16 bits and 32 bits integer *constant* is associated with a global symbol with the type of the corresponding structure, and initialized with a corresponding value. Then, every occurrence of the constant is replaced by the symbol. There are two exceptions:
    1. integer literals used to specify the size of an array are left unchanged,
    2. integers used to initialize global variables are directly replaced by a structure value.
- Every operation or cast on 16 or 32 bits integers is replaced by a function call on arguments and return value of type `_int16` or `_int32`. These functions are predefined to compute an equivalent result.

After this transformation has been performed, we have a C program but not a Clight one as functions in Clight cannot return structures as a result. For instance, after transforming a function returning a 32 bits integer we obtain a function returning a structure `_int32`. To cope with this problem, we pass the result of the transformation to the CIL tool which compiles C programs into Clight ones. Concretely, what CIL does is to add an extra argument to the function which is an address where the callee stores the result. There is one exception that concerns the `main` function in the case it returns a 16 or 32 bits integer. After the transformation, it will only return the 8 least significant bits of the result. The overall transformation is described in the diagram below where the leftmost Clight is intended to be a full Clight, (with 16 and 32 bits integers), whilst the rightmost Clight only uses 8 bits integers.

$$\mathsf{C} \xrightarrow{\mathsf{CIL}} \mathsf{Clight} \xrightarrow{\mathsf{Clight32\ to\ Clight8}} \mathsf{C} \xrightarrow{\mathsf{CIL}} \mathsf{Clight}$$

Currently, some work remains to be done to integrate this transformation in the CerCo compiler (see section 4.3).

3.2 Cminor

As explained in section 2.2, we need to distinguish between (8 bits) integers and pointers (addresses) because they do not have the same size. To this end, we identify in the abstract syntax the formal parameters and the local variables that are pointers. Following this type refinement, we need to duplicate certain operations. For instance, the addition operation is refined into an integer addition and an addition of an offset to a pointer.

3.3 RTLAbs

The changes in RTLAbs also concern the difference between integers and pointers. Since variables are pseudo-registers in RTLAbs, a pointer will simply be associated with two pseudo-registers. Instructions and operations that operated on a pseudo-register now operate on a list of pseudo-registers. This list can either be reduced to one pseudo-register in the case of an integer value, or to two pseudo-registers in the case of a pointer.

3.4 RTL

The syntax of the RTL language has been revised to mirror the 8051 instruction set. An operation over an address is expanded on operations over its most and least significant bits. Load and store operations take addresses on the form of two pseudo-registers, which also is the case for assigning the address associated to a constant symbol.

3.5 ERTL

In this language we explicit the calling conventions of the target architecture. As already mentioned, the 8051 processor does not have standard calling conventions, so that we decided to adapt those of the Mips processor.

- The seventh and eighth registers of the first bank are used for the least significant and the most significant bits of the stack pointer, respectively.
- If the result of a function is an 8 bits integer then the value is stored in the `DPL` register. On the other hand if the result is an address, the value is stored in `DPTR` with the most significant bits in `DPH` and least significant bits in `DPL`, respectively.

- There is no global pointer. Global variables are stored from address 0 in the external RAM.
- The third bank of registers is used for callee-saved registers.
- The fourth bank of registers is used for parameter registers. If a function has more parameters than there are parameter registers, the remaining ones are passed on the stack.

3.6 LTL, LIN

These languages have been adapted to the 8051 instruction set.

3.7 8051

The formalization of the 8051 assembly language is described in deliverable D4.1 [4]. In the end, the untrusted CerCo compiler produces a textual assembly file that can be simulated using the `mcu8051ide` emulator.[4] Also, the compiler produces a new C program which corresponds to the source program with cost annotations.

3.8 Memory model

The interpreters of all the languages from Clight to LIN share the same memory model (the one for the 8051 has been provided in deliverable D4.1). This memory model must be adapted when porting the compiler from Mips to the 8051. In the Mips compiler, an address value is represented as a base block and an offset in this block. We keep this convention in the 8051 compiler but we must cope with the fact that an address does not fit a single memory cell.

To start with, we must be able to *break* an address in two parts in order to put each part in different memory cells. For this purpose, we introduce two new constructors for values: `Val_ptrh` and `Val_ptrl` that both have a block and an offset as arguments and represent the most and least significant bits of an address, respectively. When breaking a value address `Val_ptr` (b, off) of block b and offset off we start by computing the most and least significant bits of off, say offh and offl. Then the broken values are `Val_ptrh`(b,offh) and `Val_ptrl`(b,offl).

We also need to build an address from two chunks of addresses that are stored in two memory cells. This *merge* operation only works if the first argument is a constructor `Val_ptrh` and the second argument is a constructor `Val_ptrl` and they both reference the same block. Then, merging `Val_ptrh`(b,offh) and `Val_ptrl`(b,offl) gives the address `Val_ptr`$(b, 256 \times \mathit{offh} + \mathit{offl})$.

3.9 A short user guide

The CerCo's sources can be compiled by invoking the `make` command. The result of the compilation is the executable named `acc`. It comes with options that can be listed with the `-help` argument on the command line. For instance, when simply running the `acc` program on a source file that contains a C program, a new file is created in the same directory as the source file, with suffix `.s`, that contains the textual assembly program resulting from the

[4] `http://mcu8051ide.sourceforge.net/`

compilation. This file is compliant with the syntax of the `mcu8051bide` emulator, that can be used to simulate the execution of the assembly. When adding the option `-a` to the command line, another file is output (with extension `.c`) which contains the source program with the cost annotations.

Note that the produced assembly code makes use of an external memory and that the usage of such memory is not the default option in the `mcu8051ide` emulator. In order to enable this option, click on the `Project` menu, and then on `Edit project`. There is a box to enable `External RAM (XDATA)` and a scrolling bar to specify its size (we suggest to use the maximum possible). Also, since the produced code might be too big for standard memory, it is recommended to enable `External ROM/FLASH (XCODE)` to its maximum size. We refer to the delivered CerCo compiler for more informations.

4 Restrictions and Planned Work

This section presents the current restrictions of the compiler and the planned work.

4.1 Integration with deliverable D4.1 [4]

The 8051 specification has been recently provided as deliverable 4.1 [4]. We are still in the process of integrating some feature of the 8051 model such as manufacturer timing information into the untrusted CerCo compiler.

4.2 Integration with deliverable D3.1 [3]

Both open source [1] and commercial compilers from C to 8051, introduce non-standard extensions of the C language which consist of directives specifying the storage location of data structures (internal RAM, external RAM, etc). The formal semantics of such extensions is a problem in itself which has been recently addressed in deliverable 3.1 [3]. Depending on the available manpower, the implementation of some of these directives in the untrusted CerCo compiler will be considered.

4.3 Support for 16 and 32 bits integers

The Clight32 to Clight8 transformation still needs to be tested and integrated within the CerCo compiler. We will rely on the open source sdcc compiler [1] to build a library of functions implementing the 16 and 32 bits operations.

4.4 Optimizing instruction selection

The experiments conducted so far suggest that the size of the produced code is rather large. For instance, an implementation of the bubble sort algorithm resulted in a code of size 2316 bytes with CerCo, while the same code is 497 bytes long with sdcc (a 4.7:1 ratio). Instruction selection is the compilation phase where we can hope to reduce the size of the binary code. In the current implementation, we have privileged uniformity and simplicity over optimization. For example, every conditional jump is translated into first computing the boolean value of the condition, and then branching depending on the result. Whenever possible, a better solution would be to use directly a 8051 branch instruction. While some improvement here seems possible, we are unlikely to match the performance of the sdcc compiler whose code generation phase is described by more than 7000 lines of C code. The goal of the project being to produce a certified compiler, it seems of no use to implement optimizations that we will not be able to certify within the time frame of the project.

4.5 Support for function pointers

Currently function pointers are not supported. It is unclear whether function pointers are really needed for the applications we have in mind, but in case they are, we expect that they can be implemented with a minor effort.

4.6 Towards Work Package 5

The CerCo compiler generates annotations for portions of code that are executed in constant time. Within Work Package 5 (Interfaces and Interactive Components), work is planned to produce starting from these elementary annotations a synthetic information on the execution cost of C functions. We are currently exploring the possibility of implementing this task as a *plug in* of the Frama − C verification tool.[5] Frama − C is an industrial strength tool built to specify and verify properties of C programs. As a case study, we will apply the implemented tool to the analysis of the C programs generated by a Lustre compiler.

5 Conclusion

We presented the port of the CerCo compiler as described in deliverable D2.1 [2] from the Mips to the 8051 processor. The main design decision have been:

1. A pre-processing of the Clight source to transform 16 and 32 bits integers into structures of 8 bits integers.
2. A type distinction between memory cells and addresses in the intermediate languages.
3. A complete revision of the instruction selection phase.
4. The design of a stack in external memory and the specification of the related calling conventions in order to implement the general recursion mechanism of C programs.

Following this approach we have been able to keep the general architecture of the compiler and to port without modification the labelling approach to the synthesis of cost annotations.

References

[1] *SDCC compiler user guide*, Nov. 2010. SDCC 3.0.1, revision 6041.

[2] R. M. Amadio, N. Ayache, Y. Régis-Gianas, K. Memarian, and R. Saillard. Deliverable 2.1: Compiler design and intermediate languages. Technical report, ICT Programme, July 2010. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[3] B. Campbell and R. Pollack. Deliverable 3.1: Executable formal semantics of C. Technical report, ICT Programme, Dec. 2010. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[5] `http://frama-c.com/`

[4] D. P. Mulligan and C. Sacerdoti Coen. Deliverable 4.1: Executable formal semantics of machine code. Technical report, ICT Programme, Dec. 2010. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[5] C. Steiner. *8051 Tutorial*, 2010. `http://www.8052.com/tut8051`.