A Translator from CRL2 representation of PowerPC Assembly to ALF

Deepthi Devaki A R
EURECA Exchange Student
Mälardalen University, Sweden
Home University: Amrita Viswavidyapeetham, Kollam, India
Department of Computer Science and Engineering
ddi09001@student.mdh.se
deepthidevaki@gmail.com

Supervisor:Andreas Ermedahl
Examiner:Björn Lisper

July 7, 2009

Abstract

Real Time systems are systems which must give accurate results within a precise time period. These systems have now become an indispensable aspect of our day to day lives. As the importance of real time system increases, the need to ensure safety increases.

Imagine an intelligent sentry android with decision making ability which guards a frontier. It has to make quick analysis of images in order to decide whether it has to shoot, warn or say "howdy partner”. In such cases, it is desirable to make a correct calculation (to identify friends and foes) and also to make a correct timely behaviour after analysing data from its sensors. This is done by the inbuilt software of the android. There lies the risk of the sentry robot not matching the image within a given time and failing in functionality. For safety reasons, in such real time applications, where meeting deadlines is critical, it is sought-after to obtain the worst case execution time of programs.

Worst Case Execution Time (WCET) analysis finds the upper bound on the execution time of a program. By obtaining this, we can design a system which works fine even in the most stressed state. SWEdish Execution time Tool (SWEET) is a tool for WCET analysis developed by the Mälardalen WCET research group. SWEET performs static analyses on an intermediate code format named ALF (Artist2 Language for WCET flow Analysis) and finds an upper bound of the worst case execution time. aiT is another WCET tool developed by AbsInt company for static analysis in real time system. They maintain a format known as CRL (Control flow Representation Language) to represent various types of object code formats in terms of control flow graphs.

The main objective of this thesis work is to write a translator from CRL2's representation of PowerPC assembler code to ALF.

Preface

Looking back at the six months I have spent in Sweden, right from the day I landed to the day I have finally completed my thesis, it has been a long journey to cherish.

As I reflect with my thesis done, I realize there are so many encouraging people I met along the way who have helped me directly or indirectly in this work. It is with immense gratitude that I write this page thanking all those noble souls without whose help and support, this thesis would not have seen the light of the day.

First and foremost, it is my guide, Andreas Ermedahl I would like to thank. He has been the friendliest and most approachable supervisor I have ever worked with. My weekly meetings would never seem like serious meetings and were more of discussions. He was always there to answer even when faced with minor problems. I also thank him for his patience and interest in shaping my thesis report the way it is today.

Next, it is my friend Nithya Vijay I owe my thanks to. Nithya and I have had common problems to encounter in design phase of the thesis work before delving into processor specific translation. Together, over the lunch table and lab rooms, we would discuss and devise a generic solution and overcame the setbacks. Hence most of the introductory parts in this report have been co-authored with her.

I also thank the other members of WCET team- Björn Lisper, Christer Sandberg, Linus Källberg, Jan Gustafsson and others who would respond to the mails and gave timely help.

I also want to express my gratitude to Prof Sasikumar Punnekat, Mr.Radu Dobrin and the EURECA board for the assistance and funding, which made my study in Sweden a dream come true.

And of course, not to forget is my HOD Mr.KrishnaKumar, other faculties and my friends in India, who have been supportive throughout.

And above all, it’s the blessing of AMMA-the divine power which has been with me throughout and has helped me in successful completion of this thesis.

Table of Contents

1. Introduction

1.1 Real Time systems

Real time systems are systems which must give accurate results within a precise time period. This means that the accuracy of the system depends not only on the logical correctness of the output but also on the time the output is produced. Real time systems need not to be fast, but it is required to produce a response within a specific time. It should not be too quick nor too late. Real time systems are computer systems mostly embedded in air craft controllers, nuclear power plants controllers, intelligent vehicle highway systems etc.

Hard Real Time System and Soft Real Time System

Hard real time systems are systems which operate within the confines of a stringent deadline missing which causes disastrous effects. For example, an air craft controller is a hard real time system because a system failure may lead to a catastrophe.

Consider the case of an anti-lock braking system in a car. The brakes should be released in a short time to prevent the wheels from locking. Any delay in the response of the system may lead to an accident. So it is crucial to verify the upper bound of the execution time of this task.

Soft real time systems, on the other hand can tolerate some delays. A missed deadline does not result in any disastrous events, but may cause minor inconvenience to the users. A live video system is a good example of a soft real time system. A delay in delivering of video frames may result in low quality, but the system can continue its operation.

1.2 WCET analysis

The Worst case execution time of a program is the longest time it may take to execute the program. The Best case execution time on the contrary is the shortest run time for a program.

As mentioned above, most of the hard real time systems are required to respond to events in its environment within a fixed time period. Any failure to meet this deadline may lead to catastrophic results.

WCET analysis aids the analysis of real time systems. The analysis results can be used to determine whether a particular task will meet the specified timing constraint and thus provide timing guarantee for the overall system behavior. The WCET analysis can typically be used in time critical applications like nuclear power plant controller, flight control system, anti-lock braking system, artificial pacemaker etc.

1.3 SWEET

SWEET (SWEdish Execution time Tool) is a prototype tool developed by Mälardalen University WCET research group. Sweet performs analysis like automatic flow analysis on Intermediate code, processor behavior analysis, instruction cache analysis on level one cache, pipeline analysis and also determines the upper bounds of execution time . The flow analysis determines the possible program flows or dynamic behavior of the system. SWEET performs its flow analysis on intermediate code level. This thesis work aims at supporting an intermediate code format called ALF. SWEET low level analysis supports NECV850E and ARM9. The figure depicts the SWEET tool architecture.

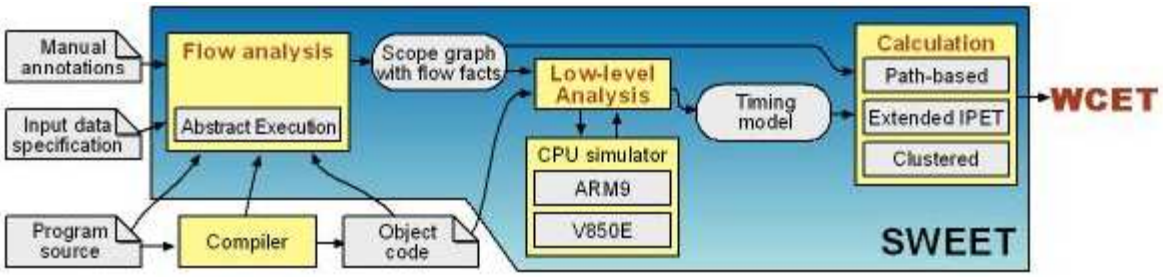

Figure 1-1 :Architecture of SWEET

1.4 aiT

aiT tool is a timing analysis tool developed by Absint Angewandte Informatik GmbH. It is used to determine the upper bound of execution time of code snippets in executables. Unlike SWEET, aiT works purely on executable. The use of executable helps to retrieve information on register usage and instruction and data addresses which are valuable for cache analysis and timing of memory accesses. The following figure shows the architecture of aiT [9].

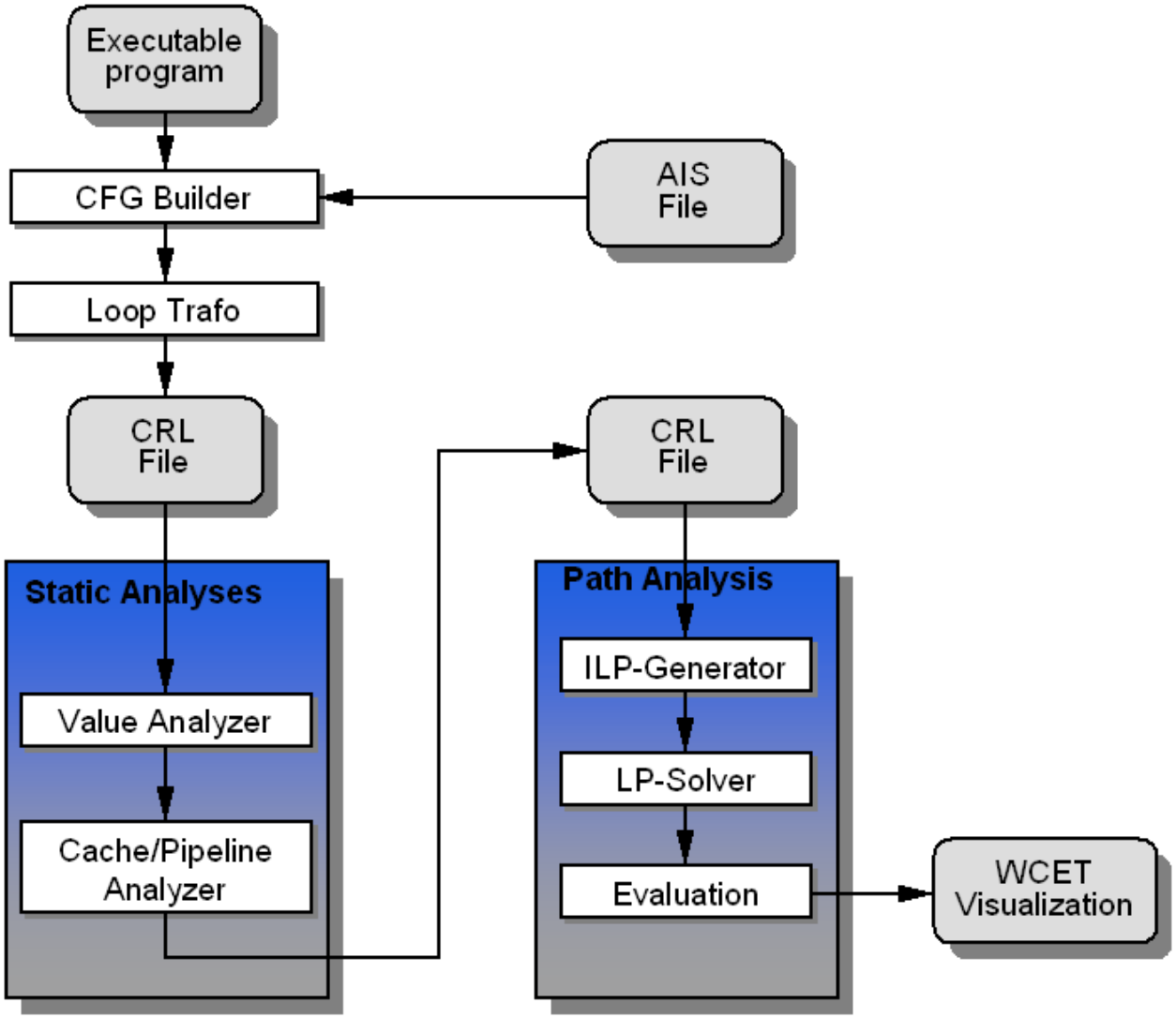

Figure 1-2: Architecture of aiT developed by AbsInt GmbH

1.5 ALF

ALF (ARTIST2 Language for Flow Analysis) is an intermediate format designed for program analysis. ALF is used as the input for SWEET for WCET analysis. ALF is designed to be possible to generate from a rich set of sources: linked binaries, source code, compiler intermediate formats, and possibly more. This has certain implications for ALF's program model, which must encompass both high- and low-level constructs while being as amenable to program analysis.

1.6 CRL2

The aiT tool uses CRL2 format for storing control flow information. aiT reconstructs the control flow information from the executable and converts it in to CRL (refer Fig 2). CRL is a human-readable intermediate format designed to simplify analysis and optimize at the executable/assembly level. CRL2 is a generic and processor independent format usable for optimization of machine code, static analysis (including WCET analysis) and assembly language. It supports the integrated representation of control flow graph and intermediate analysis results. A C/C++ library reads/writes CRL2 interface files in a text-representation format and provides an API to the CRL2's data structures used by the components of the timing-analysis tool suite.

1.7 Control Flow Graphs

The Control flow graph is a representation of flow of a program. Each node in a CFG represents a basic block. Usually there will be one CFG per procedure/function. The CFG of a procedure stores information about its instruction in basic blocks. A basic block is generally a linear piece of code without any jumps. The directed edges between two nodes represent the control flow. The control flow graph is crucial for compiler optimizations and static analysis of programs. A call graph (CG) represents possible calls between functions. In CRL2 however, the CG is encapsulated in the CFG.

1.8 Compilers and Translators

A language translator is a program which converts a program in one language to another (Alfred V. Aho). The translation can be from any level (high level, assembly code or binary) to any level. For example, a compiler is a translator which converts source code to a low level language (assembly program or binary program). Assemblers are translators whose source program is assembly program and target is in machine program.

Figure 1-3 : Role of a translator

1.9 Overview of thesis

The main objective of this thesis work is to write a translator from CRL2 Representation of PowerPC assembler code to ALF. The aiT tool reconstructs the control flow from the binary executable code, annotates it with the required information and translates it in to CRL2 code. This CRL2 format serves as the input for the translator which converts it into ALF code. SWEET tool uses this ALF code for the flow analysis.

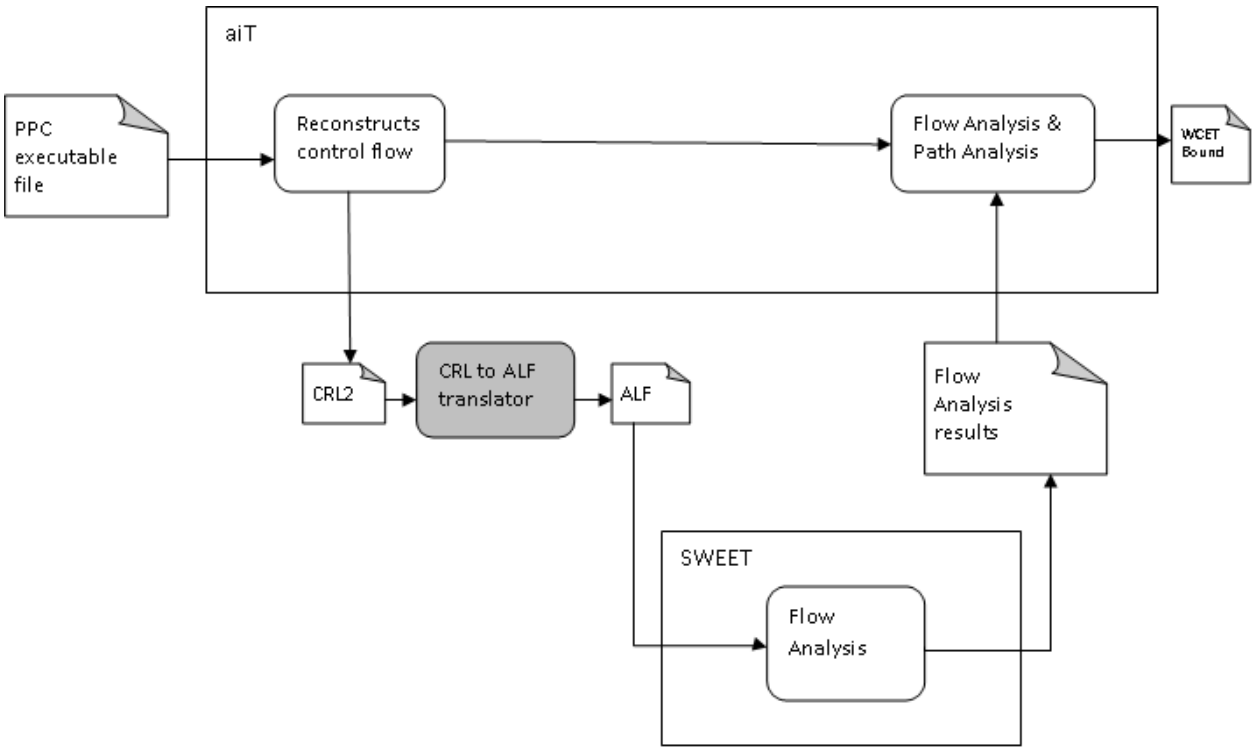

Figure 1-4: Overview of Thesis. CRL2 to ALF translator is the output of this thesis

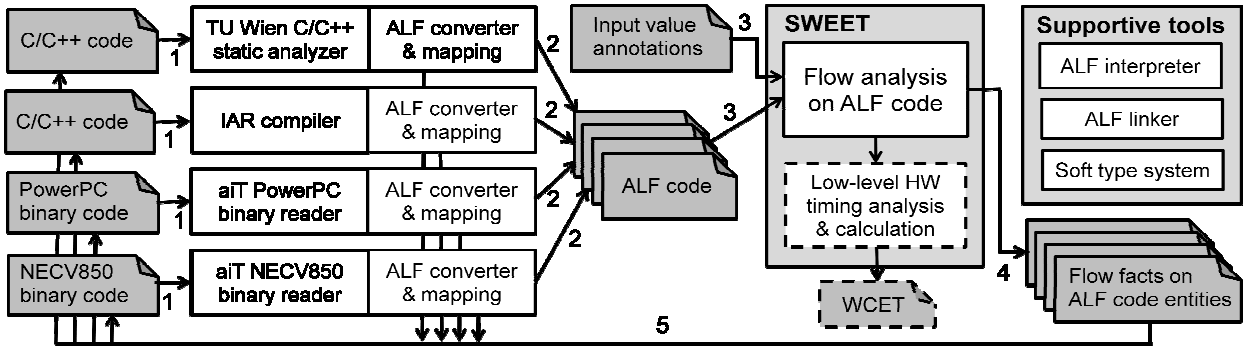

Figure 1-5: Overview of SWEET

1.10 Purpose of the Thesis

Flow analysis is a part of WCET analysis. The flow analysis gives the information about the bound on the number of times a loop iterates, which functions get called , etc. From the results of flow analysis, further study can be done on control flow information, number of iterations etc. ALF is the input format for the flow analysis of SWEET. CRL2 code is generated from the executable binary code of a program. Till date, there has been no tool to convert this CRL2 code to ALF. Here lies the significance of this thesis which translates the CRL to ALF. The generated ALF code can be fed into SWEET tool for flow analysis. Flow analysis can also be done from the ALF code generated directly from C code. The result of both analyses can be compared and provide grounds for improving the accuracy of the WCET analysis.

The aiT tool is also benefited from the flow analysis results. They can use the results for further analysis or can use them to compare their WCET analysis results.

2. ALF

ALF is a language to be used for flow analysis for WCET calculation. It is an intermediate level language which is designed for analyzability rather than code generation. The idea behind ALF is to have a generic language for WCET flow analysis, which can be generated from all of the program representations like source code, intermediate code and binary code [1].

Unlike many intermediate formats, ALF has a fully textual representation: it can thus be seen as an ordinary programming language, although it is intended to be generated by tools rather than written by hand.

2.1 Memory Model

ALF's memory model distinguishes between program and data addresses. It is essentially a memory model for relocatable, unlinked code. Program and data addresses both have a symbolic base address, and a numerical offset. Program addresses are called labels. The address spaces for code and data are disjoint. Only data can be modified: thus, self-modifying programs cannot be modeled in ALF in a direct way.

Program Model

An ALF program is a sequence of declarations, and its executable code is divided into a number of function declarations. Within each function, the program is a linear sequence of statements, with execution normally flowing from one statement to the next. Statements may be tagged with labels. ALF has jumps, which can go to dynamically calculated labels: this can be used to represent program control in low-level code. In addition ALF also has structured function calls, which are useful when representing high-level code.

Data Model

ALF's data memory is divided into frames. Each frame has a symbolic base pointer (a frame reference) and a size. A data address pointing into a frame is formed from the frame reference of the frame and an offset. The offset is a natural number in the least addressable unit (LAU) of the ALF program. The LAU is always declared: typically it is set to a byte (8 bits).Frames can be either statically or dynamically allocated.

2.2 Structure of an ALF Program

An ALF program consists of the following declarations, in the following order:

Least-addressable-unit-declaration – specification of size, in bits, of the Least Addressable Unit (for data and code memory, the underlying assumption is that they are both equal)

Endianness-declaration – specification of little/big-endianness

Export-declarations – declaration(s) of exported symbols

Import-declarations – declaration(s) of imported symbols

Allocations – allocation of static data areas

Initializations – possible initialization of static data areas

Volatile-declarations – declaration(s) of memory addresses for volatile data (which can change outside the control of the program)

Function-declarations – function (procedure) declaration(s), possibly including a ”main” procedure which then will provide the global entry point to the program

Apart from supporting flow analysis for WCET analysis tools, ALF can be used as a generic representation for different binary formats. Thus, very generic tools for analysis, manipulation, and reverse engineering of binary code may be possible to build using ALF. Possible examples include generic binary readers that reconstruct control flow graphs, and tools that can reconstruct arithmetic for long operators implemented using instruction sets for shorter operators. The latter can be very useful when performing flow analysis for binaries compiled for small embedded processors, where the original arithmetics in the source code must be implemented using an instruction set for short operators.

An Example of ALF Code

The following C code:

```
If (x > y) z = 42;
```

can be translated into the ALF code below:

```
{ switch { s_le 32 { load 32 { addr 32 { fref 32 x } {
dec_unsigned 32 0 } } }
{ load 32 { addr 32 { fref 32 y } { dec_unsigned 32 0 } } } }
{ target { dec_unsigned 1 1 }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } } } }
{ store { addr 32 { fref 32 z } { dec_unsigned 32 0 } }
with { dec_signed 32 42 } }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } }
```

The if statement is translated into a switch statement jumping to the exit label if the (negated) test becomes true (returns one). The test uses the `s_le` operator (signed less-than or equal), taking 32 bit arguments and returning a single bit (unsigned, size one). Each variable is represented by a frame of size 32 bits.

3. PowerPC Processor

PowerPC Architecture is a Reduced Instruction Set Architecture (RISC). The PowerPC architecture consists of three layers [8]. They are the following.

- User Instruction Set Architecture – Defines the basic set of instructions and registers.
- Virtual Environment Architecture - Describes the memory model for multiprocessor environment, cache control instructions and other aspects of virtual environment.
- Operating Environment Architecture - Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model.

The PowerPC supports byte (8 bits), half word (16 bits), word (32 bits) and double word (64 bits) access. The registers in PowerPC are categorized into General Purpose Registers, Floating Point Registers and Special Purpose Registers. There are thirty two 32 bit GPRs, thirty two 64bit FPRs, and many SPRs. Special Purpose Registers include Instruction Address Register, Count Register, Condition Register, Link Register, Integer Exception Register and Processor Version Register.

The PowerPC architecture defines register to register operations for most of the instructions. The source operands are provided as registers or immediate values. The load and store instructions transfer data between register and memory. There are about more than 200 instructions categorized into integer arithmetic, integer comparison, load and store, floating point arithmetic, floating point comparison, integer logical and branch instructions. One of the important concerns in this thesis is converting each of these instructions into ALF statements.

4. CRL2

CRL2 is a generic and processor independent format usable for optimization of machine code, static analysis (including WCET analysis) and assembly language. () It has a very structured approach. The outer most layer is the graph. The graph consists of several routines which can be invoked by other routines. The instructions inside a routine are structured into blocks, where each block represents a linear piece of code which follows sequential execution without any jumps or jump targets. The control flow is represented as edges between the blocks. Each instruction contains in it a specific operation and related attributes. These attributes display the mnemonic, assembly code, genname, architecture, operands and many more. The meaning of each instruction can be derived from the op_id attribute of the operation.

The following is a code snippet from a CRL2 representation of PowerPC assembly code.

```
routine r0: address=0x40fc,
gui_analysis_task=?loc("c:\\loop.apd\\a3-548-a3.ais" 11, {'0'=1}),
instruction_set="common", loop_scc=0*[], loops=1*[ /*55*/r1 ],
name="main", persistent_id=1,     section=".text",
surface_address="0x40fc" {
    block b0 (start): persistent_id=2 {
        edge e57 (linear) -> b2;
    }
    block b1 (end): persistent_id=3;
    block b2: address=0x40fc, instruction_set="common",
persistent_id=4, surface_address="0x40fc" {
        edge e58 -> b9;
        instruction i59 0x40fc:4: bytes=4*[ 0x39, 0x80, 0x00, 0x01
], file="loop.c", lines=2*[ /*22*/1, /*31*/2 ], lines_start=2*[
/*58*/1, /*67*/2 ], surface_address="0x40fc" { operation o60 "li
r12, +1": arch='UISA', assembly="addi $, $, $", dst=1*[ /*66*/'r12'
], form='D', genname='addi', op_id=0x738000000, optype=3*[ /*45*/
'GPRegAll', /*53*/'GPRegZero', /*61*/'signed' ], persistent_id
=5,src=3*[ /*24*/1='zero', /*34*/+1 ];       }
    }
    block b9 (call): address=0x40fc, loop_call=1,
persistent_id=0x14, surface_address="0x40fc" {
        edge e89 (local) -> b10;
        edge e90 (call) -> r1;
    }}
```

Some of the attributes n the above example is explained below [6].

Routine:-

`address`: This is the bus address of the instruction before address translation by MMUs. An empty routine also has an address. It is assigned conceptually by closest correspondence and is likely (but not necessary) to be equal to an address of some non-empty block.

`name`: The human readable name of the routine as extracted from e.g. an executable's symbol table.

Block:-

`address`: This is the bus address of the instruction before address translation by MMUs. Empty blocks still have an address. It is assigned conceptually by closest correspondence and is likely (but not necessary) to be equal to an address of some non-empty block.

`instructions_set` : For multiple instruction set architectures, it indicates the instruction set of this particular instruction.

`block_type`: At some nodes, this field defines the type of block. For example, a call block.

Instruction:-

`address` : This field is same as that of routine and block.

`instructions_set`: For multiple instruction set architectures, it indicates the instruction set of this particular instruction.

`width`: It shows the instruction width in bytes

Operation:-

`op_id` : It is a unique numerical identification of the operation. It can be used to map semantics to the operation in an analysis.

`genname`: A symbolic identification of the operation. It need not be unique, but it is defined to identify a conceptual class of operation, for instance, a family of 'add' instructions.

`ext` : It gives the list of extensions of an operation

`src` : It is a vector of symbol or numeric source resources of an operation.

`dst`: Destination resources of an operation.

`op` : Any operand resource of an operation is referred by this field. The indices correspond to the 'src' and 'dst' vectors. Further, in rare cases, there can be operands that exist conceptually, or are listed for consistency reasons. In these cases it may happen that only the 'op' entry exists, but neither a 'src' nor 'dst' resource is available.

`mnemonic` : It gives a human-readable textual representation of the operation

Besides the attributes explained above, blocks have edges which denote the control flow in the program. An edge has a target block or routine. The type of an edge determines the type of control flow. It can be a function call represented by a call edge, or a conditional jump represented by true edge or false edge. A true edge means that the edge is taken if the condition is true. Similarly a false edge is taken if the condition is false. Similarly there are other types of edges namely zero edge, delay edge, impasse edge, normal edge etc.

The CRL2 library is a library for exchanging the control flow graphs. It supports mixed C++ and C usage. In the implementation of CRL2 to ALF translator, CRL2 library is used for reading CRL2 code and getting the necessary information

5. Related Works

A very closely related work going in parallel was the translation of NECv850 binary executable to ALF done by Nithya Vijay. After analyzing the common features of PowerPC and NECv850, a common design for the translator was jointly undertaken.

Another related work in this field was done by Samuel Petersson in his MSc thesis work: Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS [7]. Bound-T is another WCET tool which analyses the binary executable to find the WCET. His thesis describes how he ported the Bound-T WCET tool to the Lego Mindstorms and the H8/300 processor. He made a semantic interpretation of H8/300 instructions in terms of Presburger operations. Presburger tool is used for WCET flow analysis. The thesis also describes how the resulting Bound-T version was used to analyze and derive timing bounds for selected parts of the Asterix OS. The resulting Bound-T tool version is used in real-time courses given at the School of Innovation, Design and Engineering at Mälardalen University.

A partial thesis work done by Per Wolde with the aim of making SWEET to use the AIR/CRL2 format which facilitates exchange WCET analysis results between aiT and SWEET tools can also be considered as relevant work in this regard.

No MSc thesis work has however to this date been resulting from this work.

6. Problem Formulation

To obtain the WCET flow analysis results from PowerPC assembler code, it has to be converted to ALF format as ALF and NIC are the only two formats which can be input to SWEET's flow analysis. So there arose a need to write a translator which converts CRL2's representation of PowerPC assembler code to ALF. The extracted flow information can then be used together with timing information derived by the timing analysis part of WCET tool, to derive a WCET bound.

To develop this, firstly it is required to write a code which extract information from the CRL2 PowerPC code representation.

Next, it is required to make a semantic interpretation of CRL2 PowerPC assembler code constructs in terms of ALF code and data structures. Basically, for each type of PowerPC assembler instruction or data construct found in the CRL2 format, a corresponding ALF code or data construct should be created. Moreover, some PowerPC hardware resources, such as registers and different memory areas, need to be represented by some ALF data structures.

Also, to allow SWEET's analysis results to be given back to CRL2, a mapping between the different constructs in the CRL2 and ALF formats should be maintained.

7. Problem Analysis

Before deriving a solution the following issues need to be considered.

7.1 Issues

The PowerPC CRL2 to ALF translator should translate each PowerPC instruction to corresponding ALF statement(s). While performing this all vital information required must be retained.CRL2 format contains several information like hardware architecture details, cache size, clock rate, etc which are irrelevant for flow analysis. These details can be ignored while translating into ALF.

A chief matter of concern is the effect of an instruction execution on the values/flags stored in registers and other hardware resources. The execution of one instruction may alter the value in registers such as condition register, program status word register, exception register; etc .The ALF representation is complete only if the affected registers are also represented. That is because, the execution of succeeding instruction may depend on the earlier values stored in some of these registers.

As an example, consider the PowerPC UISA instruction `beq`.

```
beq crX, target
```

The `beq` instruction checks whether the EQ bit of `crX` (condition register) is set or not. If EQ is set then control is branched to target. The EQ bit may be set by the previous compare instruction or add instruction. Hence, care must be taken to represent all the implicit updates and access of registers in ALF translation.

Some of the registers are updated by PowerPC hardware. For example, the overflow bit is set by the hardware when integer operations such as add results in an out of range value. Likewise, the exception registers are set when an exception crops up during the execution of an instruction. This exception, detected by the hardware is hard to be identified by the software. Simulation of such operations in ALF is a matter of concern.

Another important matter to be taken care of while designing the translator is the direct memory reference. Direct memory reference in a program can occur in several ways. The data memory can be directly referenced for reading data from a particular location.

For example: - `lwzx r1, r2, r3.`

The `lwzx` instruction loads the word in memory referenced by the address (`r2+r3`) into register `r1`.

The data memory can also be referenced for storing some data in a specific location.

For example: - `stwu r1, 4(r2)`

The instruction `stwu` stores the data in register r1 to the memory referred by (r2+4).

Another instance of direct memory reference is in the case of branch statements where the target address specifies an absolute or relative value. When the same code is translated into ALF the target address should pick the ALF instruction corresponding to the referred PowerPC instruction. In some cases the next instruction address is calculated with respect to current instruction address. So the correspondence of the instruction address in PowerPC CRL2 and ALF should be maintained.

An additional major concept related to memory model in a processor is the endianness. Endianness is the byte ordering for representing data in the memory.

"Little Endian" means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.) For example, a 4 byte integer

```
0X18369524
```

will be arranged in memory as follows:

```
Base Address+0   24
Base Address+1   95
Base Address+2   36
Base Address+3   18
```

"Big Endian" means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address. (The big end comes first.) The above integer would then be stored as:

```
Base Address+0   18
Base Address+1   36
Base Address+2   95
Base Address+3   24
```

The endianness plays a key role where direct memory access is concerned. There are a few instructions in PowerPC which access a single byte or half word by directly specifying the address.

For example consider the instruction `lbz r1, 1(r2).`

The `lbz` instruction loads one byte from the address (`r2+1`) into `r1`. Suppose register `r2` contains the base address specified in the above example. If memory model is little endian then `r2+1` refers to `95`. If it is big endian then `r2+1` refers to `36`. So the memory model of ALF code should be compatible with that of PowerPC assembly code.

8. Design of the Translator

Considering all the issues mentioned in the previous section, we now discuss the different strategies to design the crl2 to ALF translator.

8.1 Translation Strategy

The translation process consists of two modules.

(i) Analysis of CRL2 code
(ii) Code Generation.

The analysis module retrieves information such as instruction, operands, and control flow from the crl2 code. The Code Generation module produces ALF code using the information given by the analysis phase. The following figure illustrates the translation procedure.

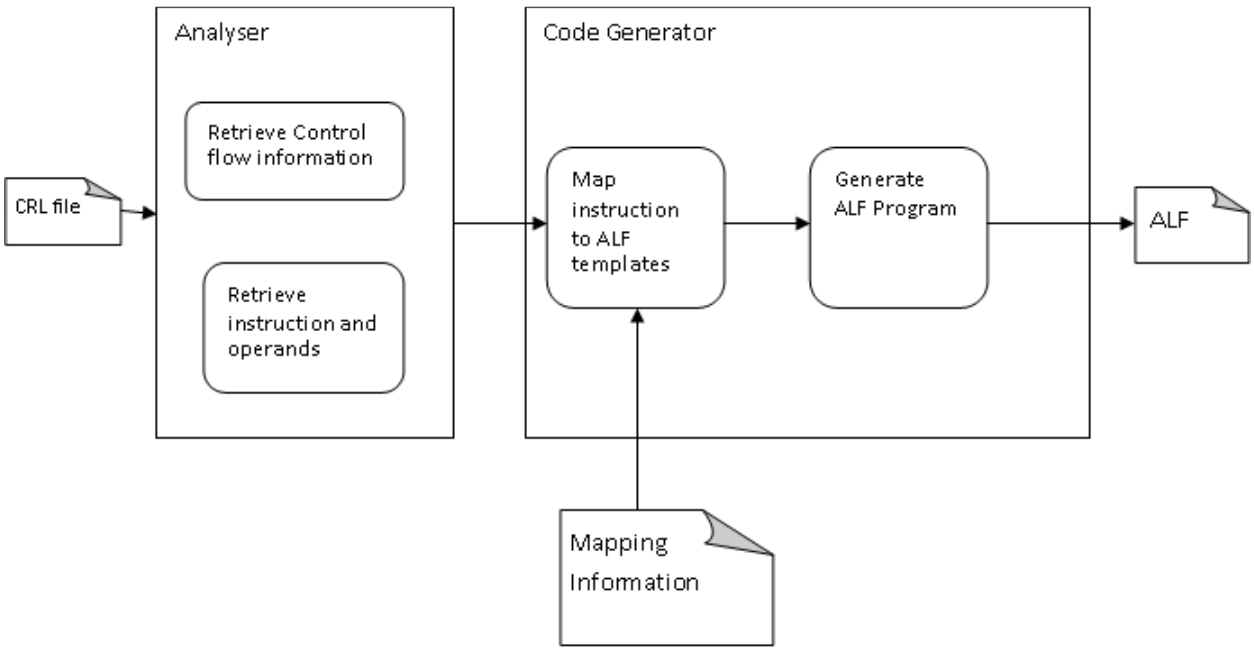

Figure 8-1: Design of CRL2 to ALF Translator

(i) Analysis

During analysis, the data describing each CRL item is passed to the Code Generation Module where these data is processed to produce the equivalent ALF structure.

To generate ALF statement for an operation, data on the semantics of the operation and operands are required. It is during analysis these information are collected. The semantics of instruction are learned from the processor manuals and are used to translate instructions to ALF code. But for a complete ALF program, the information about instructions alone will not suffice. Information about routines, blocks and edges also need to be extracted during this phase. How these extracted data is used in producing Alf statement is discussed in the subsequent sections.

(ii) Code Generation

Code Generation is responsible for producing the ALF code semantically equivalent to the CRL2 representation. The Code generation phase consists of two steps.

(i) Finding the appropriate ALF template

(ii) Converting the template to ALF code.

The data acquired during analysis is used to locate the matching ALF template. These templates, which are provided externally to the code generator, have incomplete ALF statements of the counter CRL structure. The missing elements in ALF templates are filled using the data provided by analysis module. For example, the template for an instruction will have the semantics of its operation but its operands are unknown until it is obtained from the analysis phase. The following sections discuss how each CRL item is mapped to ALF structures.

CRL Routine

Comparing the structures of both ALF and CRL2, we can see that the CRL routine and ALF functions are identical. When a routine/function is invoked from a point of execution, the control is returned to the same point after its execution. Therefore, a CRL routine can be mapped to a corresponding ALF function.

CRL Block

A CRL block represents a linear piece of code which follows a sequential execution without any jumps or jump targets. As ALF does not include any explicit

representation of a control flow graph, it does not have any structure equivalent to CRL block. However, considering the normal structure in a program, a set of instructions constitute a block. Similarly, in ALF, a set of instructions can be thought of as a block and they can be identified by giving a particular label complementing the block name.

CRL Edge

Edges are important elements in a control flow graph and are used to describe the control flow in the program. ALF does not have any equivalent structure to represent the control flow. Hence, edges cannot be directly mapped to any ALF structure. However, the information contained in a CRL edge can be effectively used to resolve the targets in branching statements.

CRL Instruction

Each CRL block comprises of several instructions which in turn contains in it a specific operation. An instruction can be matched to ALF statement(s) which does the same operation. So, it is possible to write the equivalent ALF statements by understanding the semantics of the instruction.

Structure of Mapping Table

Suitable mapping information must be available to help in matching of instructions and ALF templates. The mapping information can be provided in the following two ways.

(i) A table consisting of each assembly instruction and its corresponding ALF statement

(ii) Separate text files containing equivalent ALF template for each assembly instruction.

In the former method a separate parser for parsing the mapping table file has to be built. The parsing may take a considerable amount of execution time. As the number of instructions in the input CRL2 code increases the total time taken for parsing also magnifies. As a result, in case of large programs, the performance may be affected.

In the latter method, where a dedicated file is kept for each assembly instruction, the task of searching the mapping file is given to OS. The mapping file for an instruction is stored in a file named by its “genname”. The “genname” is a CRL2 attribute which serves as a symbolic identifier of an operation. It may or may not be same for a class of instructions.

For example, the Power PC instructions `addi, li, la, subi` are of same class and have been assigned the same genname "`addi`". Consequently the mapping file for all the four instructions is named as "`addi.map`". In NEC instructions, `add_imm` and `add_reg` are two different gennames for the same basic operation of addition.

Thus, to get the equivalent ALF statement of an assembly instruction we just have to read the corresponding text file. No separate parsing functions are required as the file handling is done by the OS. So this method is better compared to the previous one in terms of performance and reliability.

Structure of Instruction Template

The mapping file of each instruction consists of its equivalent Alf template. The value of certain objects in the ALF template depends on the input CRL2 instruction, such as operands in the instruction. Hence, some particular representation must be used to represent those dynamic data. The code generator reads the template and fills in all the missing information and produces the complete ALF statement.

8.2 Memory Model

Both the PowerPC architecture uses register-to-register operations for most computational instructions. There are load and store instructions which transfer data between memory and registers. For representing these instruction in ALF, its operands – registers and memory – also has to be modeled. The simplest and accurate way of representing registers are using the variables. In ALF data area is represented as frames. Each frame has a symbolic base pointer (frame reference or fref) and an offset. The registers of PowerPC processor can be represented using these frames, with their names as fref and the size allocated. So the General Purpose register r1 in the PowerPC processor will be represented in ALF as `fref r1` with size 32 bits.

Main memory can be represented in a similar way as registers, but with a small difference. It is accessed by its address and its size may vary. The data in the memory can be accessed in different ways. It can be accessed as byte or as a word or as a half word at a time. Consider the following example:-

Address	1000	1001	1002	1003
Data	b8	12	8c	05

Figure 8-2: Memory Layout of a 32 bit data

The above table shows one word in memory. To access as a single byte, we can use instructions like `lbz` and `lbzu` to read `b8` first, followed by`12`, then `8c` and finally `05` . By using specific instructions like `lha`, `lhz` it is possible to load half words. In that case the memory is accessed two bytes at a time. As a result, `b812` is read at a time if the operand to instruction `lha` is `1000`. Finally we have instructions to load a word at time, such as `lwz`, `lwzu` etc. In this case if the operand is `1000`, then `b8128c05` is read all together at a time.

Since it is difficult to have a frame for each byte in memory, it is efficient to have a single frame to represent the whole memory. The data in it can be accessed using its address as the offset to the frame base pointer.

The data area in ALF will look like the following

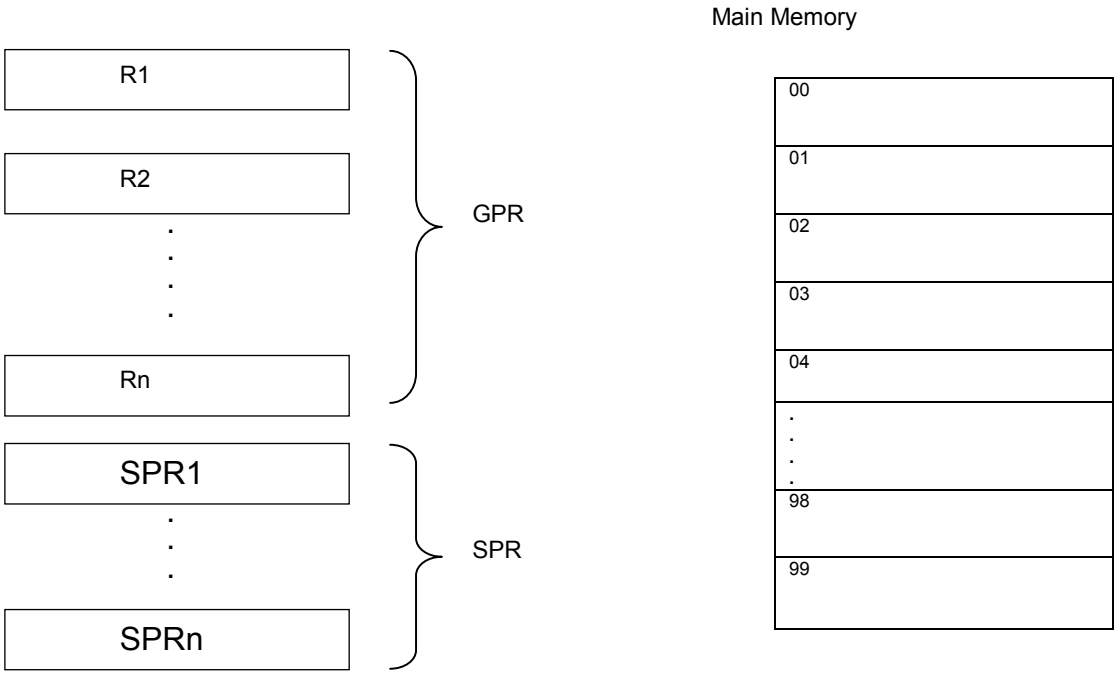

Figure 8-3 : Memory Layout in ALF

9. Solution

A working model of the design discussed in the previous section was implemented in C++. It made heavy use of the libcrl2 library for the information exchange to and from CRL2 format. Using libcrl2 library the CRL2 file is read into a CRL graph. Once the graph is read, we can write code that traverses through each routine, block, instruction, and operation in CRL2 to generate the equivalent ALF statements or operations.

As discussed in the previous section, CRL routines are equivalent to ALF functions. So an ALF statement for a function is generated for each routine in the graph. An example crl2 routine and its equivalent ALF function is shown in the following example.

```
routine r0: address=0x40fc, gui_analysis_task=?loc("c:\\loop.apd\\a3-548-a3.ais"
11, {'0'=1}),instruction_set="common", loop_scc=0*[], loops=1*[ /*55*/r1 ], name="main",
persistent_id=1, section=".text", surface_address="0x40fc" {block b0 (start):
persistent_id=2 { ... ... //other elements of routine ... }
```

```
{ func { label 32 { lref 32 f_main } { dec_unsigned 32 0 } }
        { arg_decls }
        { scope
                { decls }
                { inits }
                { stmts
                }
        }
}
```

In the given example, the `stmts` field is filled with the ALF equivalents of blocks and instructions.

We have implemented the solution using a template of instructions. For an operation, ALF statement(s) semantically equivalent to the CRL operation is generated. Problem here is that the same CRL2 instruction should always generate the same ALF code except that the CRL2 operand (values) should be dynamically translated to corresponding ALF operand values .The equivalent ALF template is stored in a file named by the concatenation of genname and op_id of the CRL operation. Unfortunately, a single Alf statement may not always be sufficient to fully represent the semantics of the operation. Consequently, one single instruction in CRL may be converted to a set of ALF statements.

In the ALF template the operands are represented as `$src` and `$dst` so that it can be replaced by the exact values given in the CRL operation. Consider the following example.

This is CRL2 representation of add instruction in PowerPC

```
instruction i66 0x4104:4: bytes=4*[ 0x7c, 0x63, 0x62, 0x14 ],
file="loop.c", lines_single=5, lines_start_single=5,
surface_address="0x4104" {
operation o67 "add r3, r3, r12": arch='UISA', assembly="add $, $,
$", dst=1*[ /*71*/'r3' ], form='XO', genname='add',
op_id=0xf7c000214, optype=3*[ /*52*/'GPRegAll', /*60*/'GPRegAll',
/*68*/'GPRegAll' ], persistent_id=8,src=3*[ /*24*/1='r3',
/*34*/'r12' ];        }
```

The ALF template equivalent to add instruction is the following.

```
{ store
     { addr 32 { fref 32 $dst1 }  { dec_unsigned 32 0 } } with
     { add 32
     { load 32 { addr 32 { fref 32 $src1 } { dec_unsigned 32 0 }}}
     { load 32 { addr 32 { fref 32 $src2 } { dec_unsigned 32 0 }}}
     { dec_unsigned 1 0 }
     }
}
```

The translator program reads this template and replaces all the `$dst[i]` by ith string in the '`dst`' attribute of CRL2. Similarly all `$src[i]` is replaced by i th string in the '`src`' attribute. In the example mentioned above, `$dst1` is replaced by `r3` and `$src1` and `$src2` by `r3` and `r12` respectively. Before replacing the operands, the alias table is checked to see and if there is any aliasing for the operand. If there is, then its alias is placed in the template instead of the original operand. Also a label name is assigned to each instruction to aid in mapping back the ALF statement(s) to the CRL instruction. The label name is generated using the routine name, block persistent id and instruction address. Therefore the final ALF statement equivalent to the above CRL instruction will be as follows

```
{ label 32 { lref 32 label_main.L1_0x10_0x4104 }
     { dec_unsigned 32 0 } }
{ store { addr 32 { fref 32 r3 } { dec_unsigned 32 0 } } with
{ add 32 { load 32 { addr 32 { fref 32 r3 } { dec_unsigned 32 0 }
} }
{ load 32 { addr 32 { fref 32 r12 } { dec_unsigned 32 0 } } }
          { dec_unsigned 1 0 } } }
```

There are cases in which a CRL2 statement must be realized by a set of ALF statements which include jumps or branches within it. In such cases, a label name is generated to denote the branch location. Every label in the template is represented as $label_<labelname> where labelname can be any string. To overcome the label name conflicts which may arise when the instruction repeats in the same routine, a distinct label count is appended to the label name. This makes the label name unique.

Besides the instructions, the CRL blocks also have details of edges in them. The information on edges is used to resolve the targets of the branch instructions. For each edge type, the equivalent ALF branch statement is generated. For a call edge, a

function call statement is produced which calls the target function of the call edge. This is shown in the following example.

```
block b9 (call): address=0x40fc, loop_call=1, persistent_id=0x14,
surface_address="0x40fc" {
          edge e89 (local) -> b10;
          edge e90 (call) -> r1;
                 }
```

The edge e90 is transformed into an ALF call statement as follows.

```
{ call { label 32 { lref 32 f_main.L1 } { dec_unsigned 32 0 } } result }
```

Here ‘main.L1’ is the name of the routine in crl2.

The other type of edges found when a conditional jump statement occurs is the true edge and the false edge. The true edge corresponds to the path followed when the condition evaluates to true and the false edge corresponds to the path taken when the condition evaluates to false. A true or false edge follows a conditional branch instruction in the code. The ALF statement for this branch instruction evaluates the condition and stores the result in a flag variable. When a true edge or false edge arises, an ALF statement is generated which checks this flag and does the branching accordingly.

The other types of edges include: normal, local, zero, delay, impasse and return edges. An unconditional jump statement to its target is generated in cases of the normal, local, zero, impasse and delay edges. The same holds here i.e., the targets of branch/jump statements can depend on the value of an expression. Thus the branch information is actually analysis results and not the code itself.

These features discussed above constitute the core of the solution.

10. Results

The result of this thesis is a working translator that can convert CRL2 representation of NECv850 into ALF code.

11. Recommendation and Future Work

This project is still not fully complete. There is much to do and many challenges to be overcome.

Firstly, the translation from CRL2 to ALF has not yet been fully implemented and validated. There are many register bits and flags to be set but were ignored, as it was irrelevant with respect to flow analysis. But in circumstances where running of ALF program is important, the translator generated output program is not guaranteed to work. The translation can be thought to be complete only if the complete semantics of the CRL2 code is reflected in the ALF version including registers, flags etc.

Secondly, though a generic solution has been devised, it is currently tested for the CRL representation of the PowerPC and assemblers only. Thus the solution has not been tested upon other binary codes from other platforms.

Thirdly, an effort has to be made to maintain the same control structures in ALF as in that of CRL2.

I believe that the above mentioned reforms could be implemented in the translator as future work.

12. Summary and Conclusion

The current version of the WCET analysis tool SWEET analyzes the intermediate format of the NIC research compiler. Since the NIC complier only supports one target platform it severely restricts SWEETs usage. After the completion of the presented thesis works; it will be possible for SWEET to reform WCET flow analysis upon binary codes thus making SWEET much more easily portable to different target platforms. The translator developed can convert the binary executable into ALF which can be input into SWEET.

13. Bibliography

[1] *ALF – A Language for WCET Flow Analysis.* Jan Gustafsson, Andreas Ermedahl, Bjorn Lisper,Christer Sandberg, and Linus Kallberg. 2009**.** 2009.

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**.** *Compilers: Principles, Techniques and Tools.*

[3] Andreas Ermedahl, Jan Gustafsson, and Bjorn Lisper**.** *ALF (ARTIST2 Language for Flow Analysis) Specification.* s.l. : Dept. of Computer Science and Electronics, M¨alardalen University, Västerås, Sweden. ARTICLE VERSION: Id : articlemain.tex, v1.302009/01/0122 : 33 : 32blrExp.

[4] CRL Library. [Online] AbsInt Anganwadle Informatik. http://www.absint.com/artist2/doc/crl2/html/.

[5] CRL2 Basic Attributes. [Online] AbsInt Angenwandle Informatik. http://www.absint.com/artist2/doc/crl2/html/doc_attributes.html.

[6] Ermedahl, Andreas. 2003**.** *A Modular tool Architecture for Worst-Case Execution Time Analysis.* 2003.

[7] Petterson, Samuel. 2005**.** *Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS.* 2005.

[8] 1999**.** RCPU, RISC Central Processing Unit Refernce Manual. s.l. : Motorola, 1999.

[9] Reinhold Heckmann, Christian Ferdinand - AbsInt Angewandte Informatik GmbH**.** Worst-Case Execution Time Prediction by Static Program Analysis. *AbsInt.* [Online] AbsInt. http://www.absint.com.

[10] *The worst-case execution-time problem—overview of methods and survey of tools,.* 2008**.** 2008, ACM Transactions on Embedded Computing Systems (TECS), vol 7, nr 3, p1---53, ACM .

[11] WCET Project/SWEET. *WCET .* [Online] http://www.mrtc.mdh.se/projects/wcet/sweet.html .

[12] Vijay, Nithya. 2009**.** *A translator from CRL2 Representation of NECv850E to ALF.* 2009.