Extracting Safe and Precise Control Flow from Binaries *

Henrik Theiling

Universität des Saarlandes and AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany,

theiling@cs.uni-sb.de

Abstract

As a starting point for static program analysis a control flow graph (CFG) is needed. If only the binary executable is available, this CFG has to be reconstructed from sequences of instructions.

The usual way to do this is a top-down approach: the executable's information about routines is used to split the sequence into routines, and then, each instruction is analysed for branch targets in order to compute basic block boundaries.

When analysing safety critical real-time systems, safe and precise results are needed. The CFG the analyses traverse has to satisfy the same safety and precision requirements, because the analyses inherit all deficiencies.

In this paper a bottom-up approach for CFG approximation is presented. It starts at a set of entry points and clusters the sequence of instructions into larger units like blocks and routines. By this the algorithm is able to account for uncertainties early to generate a safe CFG.

1 Introduction

An important characteristic of hard real-time systems is that they are subject to timing constraints that determine when actions have to be taken. A failure of a safety critical real-time system can lead to considerable damage or even a loss of lives. Therefore, a schedulability analysis has to be performed in order to guarantee that all timing constraints will be met. All existing techniques for schedulability analysis require the WCET of each task in the system to be known. Since these are in general not computable, estimations of the WCET have to be calculated. These estimations have to be safe, i.e., they may never underestimate the real execution time, and they should be tight, i.e., the overestimation should be as small as possible.

Work about WCET prediction usually starts with the assumption that the CFG of the program is available. However, there are cases where only the executable program is available, from which the CFG has to be reconstructed. When analysing real-time systems, this CFG has to be safe and precise for analyses to be, too. In our research project USES we have developed a generic tool that reads executable files, analyses machine instructions and reconstructs the CFG for analyses of real-time systems.

The exact reconstruction of a CFG from the binary for modern architectures is difficult. One problem are memory indirections (jump or call tables, procedure variables, etc), guarded code and ambiguous usage of machine instructions. A constant propagation would be helpful to get precise control flow. However, for such an analysis, a CFG would be needed, so there is a chicken and egg problem. Our approach is similar to that of [20], namely to compute an approximation of a conservative CFG first, which is annotated with the aspects of uncertainty, and then to perform constant propagation on that CFG to resolve the uncertainties left.

The work presented here focuses on the first step of the CFG approximation, namely the approximation of a conservative CFG by using knowledge about the compiler and the target architecture. We construct the approximative CFG with the real-time systems analysis in mind, i.e., it must be *safe* and must be *as precise as possible*. The next steps, refining the CFG by constant propagation, loop reconstruction, and then performing analyses has been discussed in literature (e.g. in [6, 10, 13, 14, 15, 19, 20, 22]).

A bottom-up algorithm will be presented that overcomes some deficiencies of top-down algorithms. Top-down algorithms rely on information about routine boundaries from executables. These are usually given as start and end addresses. It is a common compiler technique, however, to store certain data in code sections, e.g. target addresses of switch tables. When searching for target addresses by looking at each instruction of a routine separated by the above method, the data portions might be misinterpreted as instructions and incorrect target addresses might be used for control flow reconstruction.

Another aspect is that routines might be interlocked if

*The work of the USES group (University of the Saarland Embedded Systems group) is partially supported by Transferbereich 14 of the Deutsche Forschungsgemeinschaft.

a cache-aware compiler is used. This can also not be described by using start and end addresses.

A bottom-up approach does not suffer from these deficiencies. Each instruction *must* be classified completely and correctly before it is used in analyses. Based on these safe classifications, jump and call targets are used to compute routines and their extents. This means that data portions will be skipped and interlocked routines can be handled correctly. Finally, a bottom-up algorithm does not rely on additional information from the executable. Instead, if available, it can be used for consistency checks.

Our approach aims at retargetability, so it was designed to be generic wrt. to the underlying architecture, the used compiler and the input format of the executable. This was achieved by a module concept allowing extensions.

For two selected architectures, it will be shown that the CFG approximation algorithm is able to reconstruct the total CFG without the need of a constant propagation.

1.1 Overview

This paper is structured as follows. Section 2 will identify the problems CFG reconstruction faces. Section 3 will formalise the bottom-up approach to CFG reconstruction. Section 4 will present the module interface, Section 5 will show how our generic algorithms work and how they solve the problems. Section 6 will give an overview of the modules and describe examples for given architectures. In Section 7 we will evaluate our algorithms in practical tests, Section 8 outlines future work. Related work will be discussed in Section 9. In Section 10 a conclusion will be drawn.

2 Control Flow Reconstruction

A program's inter-procedural control flow graph (ICFG) is split into two parts.

1. A *call graph* (CG) describes the relationship between *routines*[1]. Its nodes represent routines, its edges represent routine calls and returns.
2. Each routine has a *control flow graph* (CFG), describing jumps in the routine. Its nodes are called *basic blocks*. These are sequences of instructions that are entered at their first instruction and executed without the possibility of control flow exiting or entering except for after the last instruction. The edges in the CFG describe jumps and fall-through edges.[2]

[1]Sometimes called 'functions' or 'procedures'

```
movh      d11, 45057
addi      d11, d11, -14584
ld.a      a15, [a12]0x0
ld.bu     d9, [a15]0x0
add       d15, d9, -43
mov       d1, 58
jlt.u     d1, d15, L2
mov.a     a3, d11
addsc.a   a15, a3, d15, 2
ld.a      a15, [a15]0x0
ji        a15
```

```
jge.u     d9, 5, L1
movh.a    a15, 45057
lea       a15, [a15]-0x3a44
addsc.a   a15, a15, d9, 2
ld.a      a15, [a15]0x0
ji        a15
```

Figure 1: Examples for switch table code generated by the HighTec GNU C compiler for TriCore. The aspects that have to be extracted (the switch table base address and its size) are in bold. Instructions that need not be considered are in italic. They were inserted by the optimising compiler's instruction scheduler.

The CG-edges and CFG-edges will be called *branches*.

The input of a CFG approximation is a binary executable, object code, or an assembly file. These contain a series of instructions, coded as bytes or text. Additional information may (but need not) be available: entry points, label and routine addresses, label and routine names, relocation tables, etc.

2.1 Problems of ICFG Reconstruction

1. Branch targets have to be determined. This might not always be possible for computed branches (Figure 1).

   Examples are switch tables, exception handling code, etc. The tool for CFG approximation should resolve as many of them as possible.
2. Delay slots complicate the computation of the basic block boundaries.
3. Instruction sets may have instructions that are ambiguous wrt. how program flow is controlled.

   E.g. it might be complicated to find the end of a routine, because the given target does not have a dedicated 'return' instruction (Figure 2).
4. Guarded code makes analysis complicated.
5. On architectures with very long instruction words (VLIW), instructions consist of several operations, of which more than one may be branches. The semantics of this situation differs from target to target.

[2]Fall-through edges point to immediate successors, i.e. those that are reached without a jump, but simply by the serial execution of the machine instructions.

```
cmpli   0, 0, r12, 0x14        mflr    r0
bgt     L4                     stw     r0, 0x4(r1)
lis     r10, 0x10000@h         stwu    r1, -0x10(r1)
slwi    r9, r12, 2             addi    r11, r1, 0x10
ori     r10, r10, 0xd44        ...
lwzx    r9, r9, r10            lwz     r0, 0x14(r1)
add     r9, r9, r10            mtlr    r0
mtlr    r9                     ori     r1, r11, 0x0
blr                            blr
```

Figure 2: On the PowerPC ([7]), `blr` is used for switch tables (on the left) as well as for routine exits (typical routine prologue and epilogue on the right). Routine exits are not necessarily at the end of the routine.

6. Object code and linked binaries are analysed, so multiple entry points and external routines must be handled.
7. Procedures might be interlocked or overlapping due to optimising compilers or hand-written assembly.
8. Data portions might be contained inside code blocks.

If a branch target is unknown, that branch is marked so subsequent analyses can decide whether to transform the CFG to contain *chaos nodes* (*see* [20]).

The CFG is approximated in two steps. First, a conservative ICFG is produced, then a static analysis is performed to refine it, like constant propagation by abstract interpretation. The precise *intra-procedural control flow graph* (PCFG) must be known, since if any CFG-edge is missing, it might jump *anywhere* and influence analysed properties at *any* point of the program. Assuming well-formed compiler output assembler, it may be possible to restrict the scope of interference to one routine.

If a table of all labels is available, these are the only possible targets of unpredictable jumps.

Fortunately, uncertain or missing inter-procedural edges are easier to handle (they cannot split basic blocks). Subsequent analysis steps can assume worst case without invalidating the results at other points of the program. Knowing calling conventions (e.g. *callee save registers*), it may then be possible to perform intra-procedural analyses without interference.

3 Theory

In the following, the bottom-up approach of ICFG computation will be formalised. Technical details are suspended until subsequent sections.

Let f,g be functions. $f \circ g$ denotes functional composition: $(f \circ g)(x) = f(g(x))$. Let M be a set. $|M|$ is the cardinality of M, $\mathfrak{P}(\mathrm{M})$ the power set of M.

3.1 Bottom-Up Approach

Let $M = (I, c, j)$ be a machine description with I the instruction set, $c : I \to \mathfrak{P}(\mathbb{N}_0)$ the mapping of instructions to their call target addresses, $j : I \to \mathfrak{P}(\mathbb{N}_0)$ the mapping of instructions to their target addresses and the address of the immediately following instruction if that is reachable.

It is assumed that c and j can be computed from $i \in I$ alone, i.e. an instruction contains all information about target addresses without further memory lookup.

Let $P(M) = (A, e, s)$ be a binary of M, with $A \subseteq \mathbb{N}_0$, $|A| < \infty$ the finite set of addresses that contain instructions, $e \in A$ the program entry address, $s : A \to I$ the mapping from addresses to instructions, i.e. the memory contents.

We assume that no jumps or calls leave the executable, so we restrict j and c to $I \to \mathfrak{P}(\mathrm{A})$. Further, j and c shall be omniscient wrt. computed branch targets. The next sections will be more general. It is no problem to extend the construction to sets of entry points to analyse object files.

To describe the ICFG, the following must be known:

- the set $R \subseteq A$ of routine entry addresses,
- for each $r \in R$, the set $b(r) \subseteq A$ of all addresses of instructions belonging to the routine starting at r.

By $j \circ s$ and $b(r)$, each routine's CFG is fully described.

The CG is described by c and R, as $c \circ s$ describes all CG edges when applied to each $b \in b(r), r \in R$.

The series (R_n, b_n) will be used to defined R and b.

The first routine starts at the entry point: $R_0 := \{e\}$. Let

$$\begin{aligned} b_0 : R_0 &\to A \\ r &\mapsto \{r\} \end{aligned} \tag{1}$$

For R_{k+1}, $c \circ s$ finds routine entries from $b_k(R_k)$:

$$R_{k+1} := R_k \cup \bigcup_{r \in R_k} \bigcup_{b \in b_k(r)} c(s(b)) \tag{2}$$

For the routines already known, edges are found by $j \circ s$, and for new ones, the routine entry address is used:

$$\begin{aligned} b_{k+1} : R_{k+1} &\to A \\ r &\mapsto \begin{cases} b_k(r) \cup \bigcup_{b \in b_k(r)} j(s(b)) & \text{if } r \in R_k \\ \{r\} & \text{else} \end{cases} \end{aligned} \tag{3}$$

For some k, $(R_{k+1}, b_{k+1}) = (R_k, b_k)$, as $|A| < \infty$ and addresses are never removed. So $R := R_k$ and $b := b_k$.

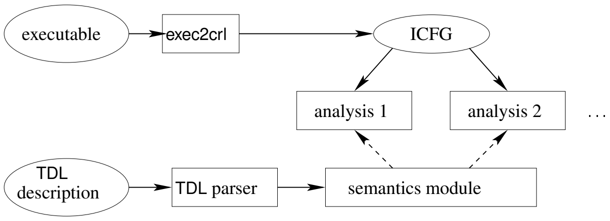

Figure 3: The structure of our analysis framework. The part discussed here is exec2crl. exec2crl provides the analyses with a control flow graph, TDL provides semantics of machine instructions. Some analyses (e.g. the instruction cache analysis) might not need TDL.

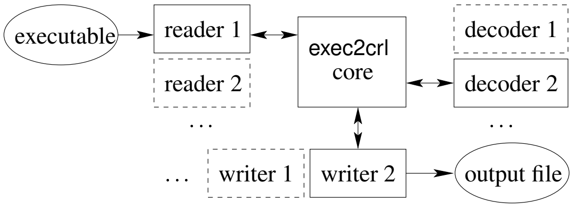

Figure 4: Module concept and communication of exec2crl. The currently selected modules are framed with a solid line, the unselected ones with a dashed line. The core implements the CFG approximation algorithms, generic code slicing and generic machine code pattern matching functionality.

4 The Modules

Figure 3 shows the our framework's structure. exec2crl approximates the ICFG. Figure 4 depicts its modules.

TDL (Target Description Language, *see* [8, 9]) will be used throughout our framework as a description language for hardware resources and the semantics of operations of the target. Only the TDL specifications and the decoder plug-ins for exec2crl are architecture dependent, so a generic analysis framework is created.

CRL files (Control flow Representation Language, [5, 11]) store the ICFG target independently. CRL consists of routines, containing basic blocks, containing instructions, containing operations. Each structure can store additional information in *attributes*, e.g. to mark unpredictability.

ICFG reconstruction is generic to work for many targets. Modules wrap target and file format dependent parts.

Reader Modules *Readers* split streams into code and data, extract entry points and function and label names, useful for human users. Henceforth, assume to have binary input, though assembly can be handled, too.

Writer Modules *Writers* store ICFGs as CRL or visualisation files like Graph Description Language (*see* [16]).

```
procedure find_cfg(program_entry)
    a= new Agenda
    g= new CallGraph
    a.insert(program_entry)
    while not a.empty
        Routine r= find_routine(a.get_next)
        g.add_routine(r)
        a.insert(r.call_targets)
    return g
```

Figure 5: The outer decoding loop in pseudo code.

Decoder Modules A *decoder* exists for every architecture family, possibly parameterised for the processor and the compiler. Its main task is to classify instructions.

The following aspects exist. * means the information can be marked preliminary so postprocessing is needed.

- the instruction width is in bytes,
- the operations contained in the instruction (for non-VLIW architectures, there is exactly one operation),
- the number of delay slots of an instruction,
- the impact of the operations on control flow: whether they are normal operations, jumps, calls, or returns,*
- fall-through edges (if the next instruction is reachable),*
- jump targets, call targets,*
- implied call targets. E.g. from calls to well-known routines like atexit in C, routine entries are extracted.*

Decoders also disassemble operations to make available the operands to subsequent analyses and for generic code slicing and pattern matching provided by the framework.

5 The Core Algorithms

CFG approximation works with an outer loop that collects routines and CG edges, and an inner one finding PCFGs.

5.1 Gathering Routines

The analysis starts at the entry address. To traverse the executable it uses an *agenda* that consists of a stack of routine start addresses and a hash table marking addresses already processed to prevent multiple processing.

The outer loop (Figure 5) puts addresses of routines onto the agenda, which are determined by find_routine.

5.2 Decoding a Routine

Figure 6 shows the algorithm for the routine decoder.

- safe_successors returns all addresses that are possibly reached from the given instruction. These are all jump targets and fall-through edges. It is important to mark

```
procedure find_routine(routine_entry)
    r= new Routine
    a= new Agenda
    c= new Set of Address        // routine's call targets
    a.insert(routine_entry)
    forever
        while not a.empty
            Instruction i= D.decode_instr(a.get_next)
            r.add_instruction(i)
            c.insert(i.safe_call_targets)
            a.insert(i.safe_successors, i.delay_slots)
        if D.finalise_routine(r) = FINAL
            r.call_targets= c
            return r
        c.insert(r.revealed_call_targets)
        a.insert(r.revealed_jump_targets)
```

Figure 6: Pseudo code for find_routine. ***R*** is the reader module, ***D*** is the decoder module that are currently used.

unpredictable aspects to be preliminary for later examination by finalise_routine.

This function cannot return computed branch targets due to the limited scope of one instruction.

- delay_slots returns the addresses of delay slots. Some analyses will account for annulled delay slot instructions (e.g. on the SPARC architecture).
- safe_call_targets returns addresses of safe callees.
- finalise_routine checks and resolves uncertainties. The whole routine is analysed to decide from the context what uncertain operations really do.
- revealed_jump_targets returns computed jump targets. The compiler must be known, e.g. to find switch tables.
- revealed_call_targets does the same for call targets.

External Routines By knowing the program's address space, calls that leave it can be marked to be external.

Additional Information Information about routine boundaries, sections types (code vs. data), labels, etc., is used to check consistency of the approximated ICFG. Routine and label names are attached to the output.

5.3 Properties of the Algorithm

The problems listed in Section 2.1 are solved either directly or by providing an easily plug-ins mechanism. This section refers to the problem items.

The algorithm finds overlapping routines (2.1.7), i.e. those that share code portions, like merged function tails produced by some optimising compilers. This would not be possible by splitting byte blocks according to routine boundaries and branches given in the input program, as top-down approaches try. Overlapping routines are analysed as separate ones that use the same addresses.

The algorithm finds interlocked procedures (2.1.7), which may be produced by cache-aware compilers. No consecutive block structure is assumed by the algorithm. This cannot be done only using boundary information. Accordingly, *holes* in the code (data portions) can be handled (2.1.8).

The algorithm is prepared for switch tables and other dynamic program flow (2.1.1). The decoders may use pattern matching and code slicing to detect them.

Delay slots are handled during the ICFG approximation and made available to subsequent analyses (2.1.2).

Uncertainties are marked clearly in the ICFG. So subsequent constant propagation can disambiguate (2.1.1) and (2.1.4). If something keeps yet unresolved, analyses are provided with that fact to take it into account.

The module system solves problems by allowing architecture and compiler dependent plug-ins when needed. This holds for all problems from Section 2.1, but is especially useful for (2.1.1), (2.1.3) and (2.1.5).

For code with multiple entries, the algorithm can simply adds them to the initial agenda (2.1.6).

Worst-Case Runtime Let n be the number of routines, m the number of instructions, $T_o(n,m)$ the runtime of the outer loop (Figure 5), $T_i(m)$ the runtime of the inner loop (Figure 6) and $T_c(m)$ the runtime of ***D***.finalise_routine.

The outer loop iterates over routines, so the worst case runtime is $T_o(n,m) = n \cdot T_i(m)$. The inner loop will execute the body only for new branch targets, as a hash table marks processed addresses. At most m iterations will be performed. By using universal hashing the expected runtime per operation is $O(1)$, so $T_i(m) = O(1) \cdot m \cdot T_r(m)$.

If ***D***.finalise_routine resolves everything by iterating all instructions and looking at only a constant number of other instructions for each of them, if follows $T_r(m) = O(m)$. This is the case for all implemented decoders.

So the total worst-case runtime is $T_o(n,m) = O(n \cdot m^2)$.

Expected Runtime Expected runtimes will be called $\hat{T}$. The previous paragraph makes pessimistic assumptions. The inner loop only re-iterates if uncertainties occur. Usually the number of repetitions is small compared to the number of instructions per routine. (E.g. the PowerPC decoder has one re-iteration due to ambiguous routine prologue and epilogue code, but only re-iterates if switch tables are found). So $\hat{T}_i(m) = O(1) \cdot m + c(m) \cdot T_r(m)$ where $c(m)$ is the expected number of re-iterations for m instructions. We expect $c(m) << m$.

The top-level loop's runtime should rather be $T_o(n,m) = O(1) \cdot n + \sum_{i=1}^{n} T_i(m_i)$, where m_i is the number of instructions per routine and $\sum_{i=1}^{n} m_i = m$. So in total, almost linear runtime is expected:

$$\hat{T}_o(n,m) = O(n) + O\big((1 + c(m)) \cdot m\big), \quad c(m) << m$$

Comparison to Top-Down Approach The worst case runtime of a top-down approach typically is $T(n,m) = O(n+m)$. It splits the program into routines and analyses each instruction. Our approach is slightly more complex, but it solves the problems we encountered. In practice, even large executables are processed in only a few seconds, so the difference is only theoretical.

6 Implementation

To be able to analyse large executables, data structures for storing decoded routines are not kept but freed as soon as a routine is analysed, the important aspects for the current analysis step are extracted. This keeps the analysis small as these structures tend to be large (for each operation, a fine-grained classification is generated).

Reader modules are straightforward to implement. Plug-ins to read ELF, Coff, Intel Hex, IEEE695, and Motorola S-Record files have been implemented.

Decoder modules are more interesting. The routines for disambiguating operations are usually complex. Focusing on processors for embedded systems, we completed the implementation of decoder plug-ins for the Infineon TriCore (Rider B) architecture and the PowerPC 403GCX. A plug-in for Motorola 68000 processors for compiled ADA code is available as a prototype.

6.1 PowerPC

The PowerPC has no dedicated 'return' instruction (Figure 2). Instead, a jump to the *link register* (`LR`) is performed. Subroutine calls fill `LR` with the return address. If the callee is a leaf routine not calling other subroutines, this address is not necessarily stored on the system stack but may be kept in `LR` until the function returns.

The problem is that the compiler also uses `LR` for other computed branches, making the instruction `blr` ambiguous. E.g. switch tables are implemented this way, too.

The finalise_routine function for PowerPC is implemented to analyse the `blr` instruction's context to find switch tables and disambiguate return instructions. If any additional jump targets are found, the analysis of the routine has to do another iteration.

Because the compiler we used has an instruction scheduler, the instructions performing a return or a switch are in general not adjacent. Code slicing was used to trace register usage and certain machine instruction patterns. Because the decoder provides generic information about instruction operands, the slicing algorithms are implemented in a machine independent way.

6.2 Infineon TriCore

Because the TriCore architecture has dedicated instructions for subroutine calls and returns, no complex disambiguation was necessary for them. For switch tables, the same techniques are used as with the PowerPC.

6.3 Exec2CRL

The implementation of exec2crl with all its modules currently has about 25,000 lines of C++ code. Most of the lines are used for decoding executable formats (9,000 lines), which is a simple, but lengthy task. Many lines are spent on the implementation of tools (e.g. the splicing algorithms) and the declaration of the module interface itself (around 4,000 lines).

The core algorithm has only 800 lines of C++ code due to the well-defined module interface. The implementation looks only slightly more explicit than the pseudo code implementation above.

The PowerPC decoder module has around 2500 lines of code (not including instruction opcode tables, which were automatically generated) which is also quite small. This shows that although the complete PCFG can be found, the effort needed to write a decoder plug-in is small.

The TriCore decoder module has around 2100 lines of code, also including the necessary pattern descriptions for computed branch target extraction.

7 Evaluation

To evaluate our algorithm, we used TriCore and PowerPC ELF executables, counted the number of uncertainties and examined how many of them could be resolved. Because we had real-time systems in mind, only statically linked executables where used for evaluation. However, several dynamically linked executables where screened to ensure that external routine calls are analysed correctly.

For the PowerPC architecture we used the MetaWare High C/C++ Compiler 3.50D to compile our test pro-

Program	Arch.	#instr.	#bytes	#comp.	#unres.	%recogn.
fft	TriCore	5563	17488	1	0	100.0%
avl	TriCore	5577	16786	2	0	100.0%
lloops	TriCore	14692	46644	8	1	87.5%
panel	PowerPC	360	1440	12	0	100.0%
ball	PowerPC	3163	12652	80	0	100.0%
ball*	PowerPC	13484	53932	275	2	99.3%

Figure 7: The results of our ICFG reconstruction. The program sizes, number of computed jumps and the number of unresolved computed jumps are given. In the example marked with *, all the unreachable library code was analysed, too.

grams. For the TriCore architecture, the HighTec C/C++ Compiler was used, which is based on the GNU C Compiler. We used the maximum available optimisation level.

The sizes of the executables ranged from a small program with only 300 instructions to Livermore loop benchmarks that produced an executable sizes of around 15000 instructions (45000 bytes). Additionally, we analysed the unused parts of a bigger executables (which were parts of the standard library) in order to analyse optimised library code. This totalled 54000 bytes (13000 instructions).

Figure 7 shows the results. Almost all computed branches could be predicted. The ones that were left unpredictable occurred in highly optimised library code. In the TriCore library, a frequently used address was kept in a register throughout several routines. The address of the switch table depended on it. In the PowerPC code, the targets of the computed branch were written into the jump table at run-time and depended on the caller of the routine.

8 Future Work

In the our research project, Transferbereich 14, CFGs will be reconstructed by the methods described here to perform several analyses for real-time systems. Prototypes have been implemented already for instruction cache analysis ([4]), path analysis ([21]), pipeline analysis ([17]), and value analysis ([18]). These will be adapted to our new generic approach by using the CFGs reconstructed in this paper. For several targets, additional constant propagation will be needed to refine the CFG especially for guarded code.

We expect this method to work for many other architectures as well. E.g. examination of guarded code of the Philips Trimedia processor revealed that the guard is a trivial 'true' for relevant parts of the PCFG. This applicability of our algorithm will have to be proven.

Currently, no effort was made to disambiguate computed calls. The ones that were encountered were calls to function pointers, so it is likely to be hard to find all possible call targets.

We will check our implementation with other compilers. The framework will be evaluated with the GNU C compiler for PowerPC very soon.

9 Related Work

In the Wisconsin Architectural Research Tool Set (WARTS), the control flow is reconstructed using program slices. Our early prototype tools for the SPARC architecture used their EEL library (*see* [12]), but genericity made a new approach necessary in order to cope with more complex architectures and compiler techniques.

One of the recent works concerning ICFG reconstruction is [20]. The paper addresses the next step of the ICFG reconstruction, i.e. disambiguating uncertainties by static analysis, constant propagation, and shows how it can be used to analyse DEC Alpha architecture. We focused more on the basic step, because we found it was complicated to obtain even a conservative ICFG due to ambiguous usage of machine instructions. Some problems [20] solves by an additional analysis step are solved in our bottom-up algorithm, too, because some disambiguation takes place during the reconstruction of the ICFG. We think, however, that for other architectures than the ones evaluated so far, constant propagation using abstract interpretation will be needed.

Much work was done in order to restructure CFGs of optimised code for debugging purposes (*see* [1, 2, 23]). All work either assumes that instructions are easily recognisable (e.g. Intel 80x86, *see* [2, 3]), or that a basic ICFG is already known. The reconstruction usually uses a top-down approach, assuming that a basic CFG exists.

10 Conclusion

In this paper an algorithm for ICFG approximation from binary executables was presented. It provides real-time systems analysers with a safe CFG. The algorithm is designed for modern processors, and involves concepts like uncertainty in the very low-level reconstruction from streams of bytes, thereby making it possible to generate a conservative ICFG approximation with most uncertainties disambiguated.

The bottom-up approach of ICFG reconstruction makes fine-grained instruction classifications available to the core of the algorithm, resulting in safer and more precise ICFGs than those produced by top-down algorithms. It can cope with interlocked and overlapping routines and it skips data portions even if located inside routines.

We presented our implementation of the algorithm, which follows a generic approach by using a module interface and showed that PCFG edges can be disambiguated for the TriCore and PowerPC architectures.

11 Acknowledgements

I would like to thank Reinhard Wilhelm for his support and for fruitful discussions about this work. Thanks a lot to Christian Ferdinand for many remarks and help with this paper. Thanks are also due to Daniel Kästner and Markus Löckelt for proof-reading and giving very useful hints of improvement.

References

[1] A.-R. Adl-Tabatabai and T. Gross. Source-Level Debugging of Scalar Optimized Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[2] C. Cifuentes and K. J. Gough. A New Approach to Debugging Optimized Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[3] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software—Practice and Experience*, 25(9), 1995.

[4] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität der Saarlandes, 1997.

[5] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems — The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, 1999.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 1998. Ausgewählt für Sonderband SAS'96.

[7] IBM. *PPC403GCX Embedded Controller, User's Manual*, first edition, 1997.

[8] D. Kästner. A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*, 2000.

[9] D. Kästner. TDL - A Hardware and Assembly Description Language. Technical report, Universität d. Saarlandes, 2000.

[10] S. Kim, S. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, 1996.

[11] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Universität d. Saarlandes, 1998.

[12] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[14] F. Martin. *Generation of Program Analyzers*. PhD thesis, Universität d. Saarlandes, 1999.

[15] G. Ramalingam. On Loops, Dominators, and Dominance Frontier. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[16] G. Sander. Graph layout through the vcg tool. In *Proceedings of the DIMACS International Workshop on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[17] J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999.

[18] M. Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität d. Saarlandes, 1997.

[19] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying Loops Using DJ Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6), 1996.

[20] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2000.

[21] H. Theiling. Über die Verwendung ganzzahliger linearer Programmierung zur Suche nach längsten Programmpfaden. Diploma Thesis, Universität d. Saarlandes, 1998.

[22] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[23] C. Tice and S. L. Graham. OPTVIEW: A New Approach for Examining Optimized Code. In *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering, Montreal*, 1998.