TUM

INSTITUT FÜR INFORMATIK

Analysis of executables for WCET concerns

Andrea Flexeder, Michael Petter and Helmut Seidl

TUM-I0838
Dezember 08

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-12-I0838-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Analysis of executables for WCET concerns

Andrea Flexeder, Michael Petter and Helmut Seidl

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany,
{flexeder, petter, seidl}@cs.tum.edu,
WWW home page: http://www2.cs.tum.edu/˜{flexeder,petter,seidl}

Abstract. In this paper we present an analysis of assembly code for safety-critical embedded environments. Since local and global variables are the core concepts affecting the control flow of such programs, we first concentrate on classifying memory accesses as candidates for local or global variables. This is achieved by an interprocedural analysis of affine equality relations. We present how to refine this information via a combined affine equality relation and interval analysis. Thus, we achieve a sufficiently accurate classification of memory accesses in this class of binaries. As an application, we refine WCET analyses by yielding alignment properties of memory accesses.

1 Introduction

Especially in the field of safety-critical real-time applications there arises the need to verify that all timing constraints on the considered system will be adhered. The fact that the program will meet its deadlines is as important as the functional correctness of the program. Deriving run-time guarantees for real-time systems has thus attracted more attention in the area of static program analysis. Since within real-time systems tasks have timing requirements, which have to be kept to assure correct functionality, one is interested in guaranteeing that each task finishes before its deadline. In this regard it is intended to statically determine the worst-case execution time (WCET) of a program. The only way to achieve reliable assumptions about this maximal time taking for a certain program to execute on a specific system, is that one has to analyse the executable, instead of the source code, written in a high-level programming language. Because of the lack of trust in the persons designing the program analysis, customers often only provide the executable. Additionally, the program analyser must only rely on the executable and cannot be sure that the execution of the source code by the processor really behaves as the code writer intends to, according to the principle WYSINWYX [5]. Since analysing the result of the compiler, one is not dependent on a certification of the correctness of the compiler.
Recently the gap between processor speed and memory access is growing. Thus, cache memories are applied to provide faster access to recently referenced memory regions. Despite this performance benefit, caches in real-time systems show up the drawback of complicating a decent prediction of the WCET in real-time systems, because the performance of the cache appears quite unpredictable. In order to consider the performance of the cache, there are many program analyses which aim at ranking memory references as

cache hits or misses [2,16,6,19]. The majority of address expressions occurring in real-time applications result from accesses to local variables [1]. Especially in the area of software development for avionics restrictive safety guidelines have to be kept, as e.g. the standard DO-178B [25]. In order to conform to this standard the C code, produced e.g. by SCADE [10], should hold a number of characteristics in order to support the certification process. These characteristics e.g. consist of the absence of pointers, address arithmetic and dynamic memory allocation, etc.. Thus, assembly code generated for C code which conforms to these characteristics only contains the concept of local and global variables. In this paper we present an interprocedural analysis on assembly code identifying candidates for global and local variables. Additionally, we introduce an analysis yielding alignment properties, as the alignment of memory accesses has a crucial impact on their performance. The main goal of our interprocedural analysis, implemented within the project *SuReal* [1], consists in yielding precise information about local variables and alignment properties to improve WCET analyses.

The remainder of the paper is organised as follows: The next subsection describes two approaches of executable analysis, using the call-string approach. Section 2 specifies the general set-up and the Power PC architecture our analysis builds on. The following section 3 addresses an analysis of affine register equalities. This information contributes in precisely identifying and correctly addressing possible local variables. Section 4 describes the enhancement of a conventional interval analysis on unoptimised assembly code, which relies on affine equality relations between memory addresses and processor registers, in order to obtain a meaningful result at all. The alignment analysis is specified in section 5 and finally we conclude.

Related Work To efficiently determine upper bounds for the WCET of a program [2], Wilhelm, Ferdinand et al. [11], [12], [13] describe an approach to predict the cache behaviour of machine programs for real-time systems. One crucial aspect in their approach is their so-called *value analysis* where value ranges for processor registers and address ranges for instructions accessing memory are computed for every program point and execution context. By this, indirect accesses to memory can be resolved. The existing approach of the value analysis is based on a simple interval analysis. In this process, the uncomfortable property is that the user has to provide divers details about the underlying program, e.g., the initial value of the stack pointer, the number of loop iterations for several loops, an upper bound for the call of recursive functions, etc. These assumptions about the program execution are required to generally find a and improve the precision of the result [11]. Additionally, a cache analysis is performed which classifies memory references as cache hit or miss [13], in order to use this information within the pipeline analysis [26] to specify the execution times for instruction sequences. The aim of this approach is to achieve reliable assumptions about the maximal time taking for a certain real-time application to execute on a specific system. Since within their analysis framework the call-string approach is taken and unrolling of loops and recursive functions is performed, their proposal can be considered more or less as a symbolic execution of the program.

In [3] Reps et al. describe their approach to recover information, e.g., variables and other information about heap-allocated objects from *x86 executables*. As stated in [4], their primary objective is to develop bug-detection and security vulnerability analyses working on stripped executables [5]. Within their tool *CodeSurfer/x86* [24][18] they implemented several static analysis algorithms whose combination allows the recovery of information about the contents of memory locations and the way they are manipulated by the executable. Their abstraction is called *abstract locations* [23] and gives an overapproximation of the values contained in memory regions, i.e., memory locations and processor registers. Within their so-called *RIC domain* they keep intervals and thus their assembly code analysis requires the call-string approach. A unification-based flow-insensitive algorithm [22] recovers information about variables and types, in order to refine the set of inferred abstract locations. These two analyses are run in an iterative strategy to identify more precise abstract locations. The aim of their executable analysis is to recover an *intermediate representation* from a stripped executable and additionally detecting whether the executable conforms to a standard compilation model.

2 General Set-up

This section describes the general set-up and the architecture our analyses are based on.

PPC Architecture Here, we restrict the hardware basis for our analysis of assembly code to the 32-bit *Power PC* architecture (PPC)[15], a three-address-machine — and therefore each instruction may contain maximally three operands. The PPC architecture offers 32 general purpose registers, 32 floating point registers and several additional special purpose registers. Instructions concerning floating point registers and special purpose registers are ignored, since we deem it improbable that such instructions will be used for address computation. The registers have a global scope valid for all the functions within the program (i.e. registers used for parameter passing, as return value of a function, or for handling local variables, etc.). In PPC, the registers $\mathbf{x}_{14}$ up to $\mathbf{x}_{31}$ are declared non-volatile and used for handling local variables. The most important convention concerns register $\mathbf{x}_1$ which denotes the *stack pointer*. As specified in the PPC instruction set [27], only *load* and *store* operations access and if so modify memory. In the following, a memory access is denoted by $M_m[p]$ where p is an affine address expression and m is the number of bytes of the memory operand. The instructions addressing memory can be applied to bytes, half-words, or words (4 bytes). Information about the number of bytes accessed by each instruction argument can be deduced by the name suffix of the instruction (b, h, w, e.g., *stb, stw, sth*). A matter of particular interest are the different addressing modes for memory access instructions, which are listed in the following table:

addr. mode	description	example
$(\mathbf{x}_j)$	register indirect	*lwz* $\mathbf{x}_i, 0(\mathbf{x}_j) \Rightarrow \mathbf{x}_i := M_4[\mathbf{x}_j]$
$c(\mathbf{x}_j)$	register indirect with immediate index	*lwz* $\mathbf{x}_i, c(\mathbf{x}_j) \Rightarrow \mathbf{x}_i := M_4[\mathbf{x}_j + c]$
$\mathbf{x}_j, \mathbf{x}_k$	register indirect with index	*lwzux* $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k \Rightarrow \mathbf{x}_i := M_4[\mathbf{x}_j + \mathbf{x}_k]$

In PPC, there arise three different possibilities of memory addressing: via a register $M_m[\mathbf{x}_j]$, via a register with a constant offset $M_m[\mathbf{x}_j + c]$, or via the addition of two registers $M_m[\mathbf{x}_j + \mathbf{x}_k]$.

According to the PPC-ABI [28], for every function a stack frame is installed where the necessary stack cells for the execution of the function body are allocated. This stack grows downward from high addresses. A stack frame is allocated by a fixed scheme of instructions for saving and restoring the function parameters (in PPC these are the registers $\mathbf{x}_3, \ldots, \mathbf{x}_{10}$) and the return value (in PPC these are register $r3$ and $r4$). According to the calling convention [28], these so-called non-volatile registers must be saved before and restored after a function call. This can be achieved by a function prologue and epilogue, generated by a compiler, as the following PPC assembly code illustrates.

Example 1. stack frame

```
01:  stwu   r1,-48(r1)
02:  mflr   r0
03:  stw    r0,52(r1)
04:  stw    r14,8(r1)
05:  stw    r15,12(r1)
...  // function body
10:  or     r3,r0,r0
11:  lwz    r14,8(r1)
12:  lwz    r15,12(r1)
13:  addi   r1,r1,48
14:  blr
```

Instruction 01 acquires 48 bytes memory for executing the function and sets the stack pointer register $r1$. Generally in the function prologue a stack frame is established, where a reference to the previous stack frame is saved on the top of stack (cf. instruction 03). According to the processor ABI [28], the first word of the stack frame shall always point to the previously allocated stack frame. Additionally, in the function prologue all the non-volatile registers it uses may be saved (cf. instructions $04, 05$). After having executed the function body, in the function epilogue the return value is written into register $r3$, additionally those registers saved in the prologue code have to be restored (cf. instructions $11, 12$).

Then, the current stack frame is deallocated (cf. instruction 13) – restoring the previous stack frame. Deallocation of a stack frame is accomplished either by loading the initial value of the stack pointer register, which is kept on the top of stack, or by incrementing the stack pointer by the same amount by which it was decremented in the function prologue (cf. instruction 13). Finally, control returns to the caller via *blr*-instruction.

If the assembly code adheres to the coding conventions with respect to the stack pointer register, according to the processor ABI, the value of the stack pointer before and after function execution is the same. However, as the PPC-ABI [28] suggests, there are non-standard calling conventions for register saving and restoring functions. These functions are required to be statically linked into any executable. In this case it is intended that after the execution of these saving and restoring functions the height of the stack frame of the caller is modified.

Example 2. non-standard calling conventions

```
_restgpr_14_l:
01:  lwz     r14, -76(r11)
02:  call    _restgpr_15_l

_restgpr_15_l:
03:  lwz     r15, -72(r11)
04:  call    _restgpr_16_l
      .
      .
      .
_rest_gpr_31_l:
36:  lwz     r0, 4(r11)
37:  lwz     r31, -8(r11)
38:  mtspr   lr, r0
39:  ori     r1, r11, 0
40:  blr
```

The saving and restoring instructions for the registers $r14 - r31$ are relocated into chains of system-level routines _savegpr_X_l and _restgpr_X_l. Example 2 illustrates the routines for restoring the non-volatile registers after a function call. In the epilogue of a function call, the callee first stores the initial value of the caller's stack pointer in register $r11$ before _restgpr_14_l is called. This starts the chain of instructions which restores the former registers of the caller. Finally, in the chain's end routine _restgpr_31_l the stack pointer is restored by moving the contents of $r11$ to $r1$ (cf. instruction 39). Then, control is passed back to the caller.

For our analysis of identifying local respectively global variables, first we discuss the appearance of local and global variables in assembly code on the basis of PPC assembly. The next example shows a simple C-program addressing global variable g and local variable l, with the corresponding assembly code aside.

Example 3. variables in PPC assembly

```
int g;
int main(){
  int l;
  g=3;
  l=5;
}
```

```
05: lis   r9,10
06: li    r0,3
07: stw   r0,10244(r9)
08: li    r0,5
09: stw   r0,8(r1)
```

Generally the data area for local variables of a function is reserved as a region of constant size on the stack. When a function is called this region for local variables is allocated and it is deallocated when the function returns. In this process, the stack pointer designates the top of the stack. Thus, each local variable is addressed by a *constant offset relative to the stack pointer*. As the initial value of the stack pointer is not fixed, but depends on the whole history (the place a function is called), the addresses of the local variables depend on the stack pointer. The global variables are managed in a global data section, hence their addresses stay constant for all functions they are used in (*absolute addresses*).

Thus, at machine code level, we obtain: the accesses to local variables arise indirectly as $M_m[\mathbf{x}_1 + c_1]$, the accesses to global variables directly as absolute address $M_m[c_2]$ where c_1 and c_2 denote constants and $\mathbf{x}_1$ the stack pointer. It is as well possible that the address of a local or global variable is computed via more consecutive instructions. Consider e.g. the store instruction 15 in example 4. There, memory is addressed via register $r31$ with offset 8. In order to identify if this memory access denotes a local variable, we have to take instruction 05 into account, which states that register $r31$ is set equal to the stack pointer register $r1$. For instruction 15, our analysis should yield that a local variable with offset -24 to the initial value of the stack pointer is addressed. As this example illustrates, it is not possible to identify whether a local or global variable is actually addressed or not via some kind of pattern matching. Thus, we provide a general approach for identifying variables in executables.

Program structure We restrict our framework to the analysis of the following class of assembly code. The SW standard DO-178B [25], especially applied in the development of aviation software, requires that the produced code is simple, deterministic and efficient and should require as few resources as possible in terms of memory and execution time [9]. The objective of this guidelines is ensuring that SW performs its intended function with a level of confidence in safety. In order to ease the certification process the considered C code, which is mostly generated by e.g. SCADE [10] in the area of safety-critical code development for avionics, should show the following characteristics

- no dynamic memory allocation
- no pointers/pointer arithmetics
- no recursion or unbounded loops
- no local arrays

which are important in the certification context. In summary the main semantical program components of the class of assembly code we consider are local and global variables, global arrays and control structures. Thus, in this paper we introduce an analysis of assembly code which is able to identify candidates for variables and global arrays and yields information about their contents. On this basis we want to verify if the assembly code to analyse conforms to the specified characteristics. In order to check whether there occurs a dynamic stack modification, we introduce a conceptual frame pointer. This frame pointer is referred to as *artificial stack pointer register* ($\mathbf{x}_{ASP}$) and provides a reference to the initial value of the stack pointer at the entry point of a function. Hence, it is possible to check whether the stack frame of the caller was modified by a function call and whether the stack frame was correctly deallocated at function exit. A dynamic modification may be provoked, e.g., by the C-function `alloca`, as the following example 4 illustrates.

Example 4. dynamic stack modification

```
int main(){
  int x,y;
  alloca(x);
  y=5;
}
```

```
//int main{
01: stwu    r1,-32(r1)
02: mflr    r0
03: stw     r31,28(r1)
04: stw     r0,36(r1)
05: mr      r31,r1
//alloca(x);
06: lwz     r9,12(r31)
07: addi    r9,r9,15
08: addi    r0,r9,15
09: rlwinm  r0,r0,28,4,31
10: rlwinm  r0,r0,4,0,27
11: lwz     r9,0(r1)
12: neg     r0,r0
13: stwux   r9,r1,r0
// y=5;
14: li      r0,5
15: stw     r0,8(r31)
//}
16: lwz    r11,0(r1)
17: lwz    r0,4(r11)
18: mtlr   r0
19: lwz    r31,-4(r11)
20: mr     r1,r11
21: blr
```

The stack level is increased by a dynamic reservation of memory within the stack frame of the caller of `alloca`. The function `alloca` returns a pointer to the allocated memory region which is automatically freed when the caller quits its stack frame. Consider instruction *stwux r9, r1, r0* (store word with update indexed) at address 13 where the content of $r9$ is written into the word in memory addressed by summing the contents of $r1$ and $r0$. Supplementary, the sum is placed into register $r1$. Here, the stack pointer is decreased (value of $r0$ many bytes) and the address of the previous stack frame is now stored at the word addressed by the new value of the stack pointer, i.e. on the top of stack.

Next, we want to verify that according to the coding convention of the processor ABI, on function exit each function correctly deallocates the stack frame it established on function entry. First, we recall how the stack management of a procedure works. Within the function prologue a new stack frame is established, where a region for local variables on the stack is reserved (cf. example 1). In this process the initial value of the stack pointer is stored on the top of stack. After a dynamic stack modification the initial stack pointer value is saved on the new top of stack (cf. instruction 13 in example 4). A whole sequence of dynamic stack modifications causes that the initial value of the stack pointer is stored on the new top of stack respectively after new space was allocated. Then, within the function epilogue the initial value of the stack pointer is loaded from the top of stack (cf. instruction 16 in example 4) and the previous stack frame is restored (cf. instruction 20). Thus, for the verification of a correct deallocation, we explicitly assume that a reference to the previous stack frame is always stored on top of stack. Now, in order to verify that the previous stack frame is correctly restored at function exit, even in presence of a dynamic stack modification, we must track the value saved on the top of stack. As in case of example 4 there may be no exact value for the dynamic growth of stack, and thus it is not possible to exactly address that memory cell denoting the top of stack. For this purpose, we introduce an additional register $\mathbf{x}_{TOS}$, which always denotes the top of stack within our equality relation analysis (refer to section 4). Back to example 4, now it is possible to verify within the function epilogue (instruction 20) that the current stack frame is deallocated correctly.

Then, after this analysis step, we have to verify that the top of stack always keeps the initial value of the stack pointer, i.e. that this memory cell is not overwritten within the function body. This can be achieved by considering the results of an interval analysis, providing information about the values of registers and memory locations. With this technique it can be verified that the program shows no malicious behaviour with respect to manipulating the top of stack and conforms to the coding conventions of the processor ABI.

In order to verify that no pointer arithmetic is used, we propose a so-called escape analysis, which yields the information that a stack address may be saved on the stack or passed as parameter. This escape analysis is sketched in section 3.

After having specified the structure of the programs we handle, we describe their representation.

Program representation The input to be analysed is a fully linked executable program without any information about program and data allocation. As customary in data flow analysis, the assembly-code to analyse is represented by control flow graphs. Within our analysis framework, each function is described by a finite control flow graph G_f

which consists of:

- a finite set N_f of *program points* of function f,
- a finite set $E_f \subseteq (N_f \times N_f)$ of *edges*,
- a mapping $A : E \rightarrow \mathit{Label}$ annotating each edge with an instruction,
- a unique entry point $s_f \in N_f$ of function f,
- a unique exit point $r_f \in N_f$ of f.

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of k registers the program operates on. Due to the fact that processors perform their arithmetic operations modulo a power of 2, the registers take values from the ring $\mathbb{Z}_{2^w}$.

Each edge in the control flow graph is annotated by the representation of a processor instruction. We model the concrete effect of every processor instruction within our analyses by the following simple constructs: assignment statements, function calls, and conditional branching. Note that some processor instructions may have side effects modifying additional registers. Within our framework this is modelled by consecutive assignments representing the whole effect of a processor instruction. For example, the PPC *conditional branch*-instruction may modify the *link register* for saving the effective address of the instruction following the branch, and possibly the *count register* whose value may be decremented. For a concrete example, consider instruction 14 up to 16 from example 6, which describe the C-construct $i < 100$. The semantics of all PPC instructions is put down to the following basic statements – linear assignments of the form $\mathbf{x}_i := \mathbf{x}_j \pm \mathbf{x}_k$, $\mathbf{x}_i := c \cdot \mathbf{x}_j$, affine assignments as $\mathbf{x}_i := \mathbf{x}_j \pm c$, and guards of the form $\mathbf{x}_i \,\square\, \mathbf{x}_j, \mathbf{x}_i \,\square\, c$ with $\square \in \{\geq, >, \leq, <, ==, \neq\}$ where c is a constant and $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k$ are processor registers. Here, we restrict arithmetic to addition, subtraction, and multiplication with a constant. Guards derive from those PPC instructions which handle the branching mechanism – there the conditional expression is evaluated and its result is placed in a single bit of the condition register (*compare*-instructions), subsequently the branch is performed according to this bit value of the condition register. Example 5 illustrates the realisation of conditional branching in PPC assembly for an *if-else*-sequence in C.

Example 5. conditional branching

```
 z = (x>y) ? 1 : 0;

lwz     r0,16(r1) //load x
lwz     r9,12(r1) //load y
cmpw    cr7,r0,r9 //place compare result in cr7
ble     cr7,0x08  //branch if cr7 result denotes less
li      r0,1
stw     r0,24(r1) //save 1 in tmp memory cell
b       0x10
li      r0,0
stw     r0,24(r1) //save 0 in tmp memory cell
lwz     r0,24(r1) //load from tmp memory cell
stw     r0,8(r1)  //save result in z
```

Furthermore, we encounter non-deterministic assignments of the form $\mathbf{x}_i :=?$ which represent a safe description of those instructions which cannot be handled precisely, but have an impact on the values of registers (e.g. *ecowx, eieio* ...). Arithmetic instructions as division, bit operations as, e.g., the *rotate and shift instructions* for shifting, rotating, extracting, clearing, and inserting bit fields in a general way and logical instructions (e.g *and, or, nand, xor*) are also represented by non-deterministic assignments. *Skip*-statements model those instructions our analysis cannot deal with. They are omitted here, because they have no effect on the underlying program state. Additionally, we consider memory access instructions occurring in form of $\mathbf{x}_i := M_m[p]$ or $M_m[p] := \mathbf{x}_i$ where p is an affine expression. The majority of address expressions occurring in executables of real-time applications result from accesses to local variables or one-dimensional arrays[1]. These address expressions are mostly affine expressions and thus it is sufficient to consider affine register relations, as described in the next section 3.

A program state assigns values from $\mathbb{Z}_{2^w}$ to processor registers when a certain program point is reached at program execution. A program execution is represented as a sequence of statements and leads to sequential transformations of the initial program state. Taking guards into account we even obtain a set of transformations. At a program point we consider the merge over all program executions reaching this program point. Thus, we describe sets of states reaching program points by the collecting semantics. To conclude, the following example 6 illustrates the control flow graph for the given assembly code to its left.

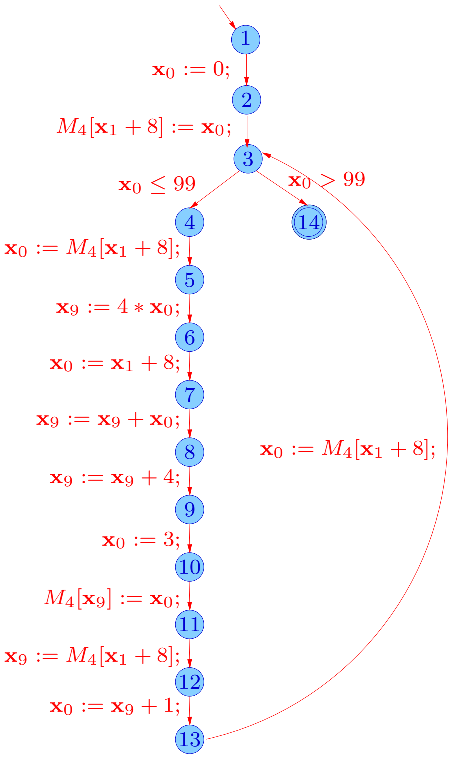
Example 6. assembly code to CFG

```
int i,a[100];
for(i=0;i<100;i++)
   a[i] = 3;
```

```
01:  li      r0, 0
02:  stw     r0,8(r1)
03:  b       0x14

04:  lwz     r0,8(r1)
05:  mulli   r9,r0,4
06:  addi    r0,r1,8
07:  add     r9,r9,r0
08:  addi    r9,r9,4
09:  li      r0,3
10:  stw     r0,0(r9)
11:  lwz     r9,8(r1)
12:  addi    r0,r9,1
13:  stw     r0,8(r1)

14:  lwz     r0,8(r1)
15:  cmpwi   cr7,r0,99
16:  ble     cr7,0x04
```


1
$\mathbf{x}_0 := 0;$
2
$M_4[\mathbf{x}_1 + 8] := \mathbf{x}_0;$
3
$\mathbf{x}_0 \leq 99$
$\mathbf{x}_0 > 99$
4
14
$\mathbf{x}_0 := M_4[\mathbf{x}_1 + 8];$
5
$\mathbf{x}_9 := 4 * \mathbf{x}_0;$
6
$\mathbf{x}_0 := \mathbf{x}_1 + 8;$
7
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_0;$
8
$\mathbf{x}_0 := M_4[\mathbf{x}_1 + 8];$
$\mathbf{x}_9 := \mathbf{x}_9 + 4;$
9
$\mathbf{x}_0 := 3;$
10
$M_4[\mathbf{x}_9] := \mathbf{x}_0;$
11
$\mathbf{x}_9 := M_4[\mathbf{x}_1 + 8];$
12
$\mathbf{x}_0 := \mathbf{x}_9 + 1;$
13

3 Affine Equality Relation Analysis

In this section, we present an interprocedural analysis inferring all valid *affine equality relations* between the processor registers. These equality relations contribute in classifying memory accesses or inferring alignment properties. Since the identification of local variables demands tracking stack pointer modifications, we extend the register set $\mathbf{X}$ by introducing the register $\mathbf{x}_{k+1}$ which represents the artificial stack pointer – for simplicity denoted by $\mathbf{x}_{ASP}$, henceforth. Thus, a state is modelled by a $(k+2)$-dimensional column *vector* $x = (1, x_1, \ldots, x_{k+1})^T \in \{1\} \times \mathbb{Z}_{2^w}^{k+1}$. According to the approach of [20] the extra 0-th component capturing the value 1 allows us modelling the semantic effects of affine assignments through linear transformations. Every linear assignment t gives rise to a state transformation $[\![\mathbf{x}_i := t]\!]x$ leading to $(1, x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots x_{k+1})$ where t is a linear expression. In the case of conditional branching, the effect is given by the set of those vectors satisfying the guard. A non-deterministic assignment $[\![\mathbf{x}_i :=?]\!]$ results in the union of assigning every possible constant $c \in \mathbb{Z}_{2^w}$ to register $\mathbf{x}_i$. Summarising, each statement induces linear transformations of the underlying program state. This specifies our collecting semantics.

Now, we abstract sets of program states by submodules of $\mathbb{Z}_{2^w}^{k+2}$. We obtain all valid affine equality relations by considering the *dual module*, i.e. all those vectors which are orthogonal to the vectors from the submodule. For self-containedness, we briefly recall a few properties of modules over $\mathbb{Z}_{2^w}$, as described in detail in [21]. $\mathbb{Z}_{2^w}$ is not a field, because only all odd elements are invertible, whereas every even element is a zero divisor. A subset of $\mathbb{Z}_{2^w}^{k+2}$ of vectors with entries in $\mathbb{Z}_{2^w}$ is a $\mathbb{Z}_{2^w}$-module iff it is closed under vector addition and scalar multiplication with elements from $\mathbb{Z}_{2^w}$. Submodules of $\mathbb{Z}_{2^w}^{k+2}$ are closed under intersection and ordered by set inclusion. A subset S of a submodule $M \subseteq \mathbb{Z}_{2^w}^{k+2}$ is called a *generator set* of M iff $M = \{\sum_{i=1}^{m} a_i s_i \mid m \geq 0, a_i \in \mathbb{Z}_{2^w}, s_i \in S\}$. $M = \langle S \rangle$ denotes that M is generated by S. The least upper bound of two submodules M_1, M_2 is given by $M_1 \sqcup M_2 = \langle M_1 \cup M_2 \rangle = \{m_1 + m_2 \mid m_i \in M_i\}$. The least element $\bot$ is $\{\mathbf{0}\}$, whereas the greatest element $\top$ is given by the whole vector space $\mathbb{Z}_{2^w}^{k+2}$. Thus, submodules of $\mathbb{Z}_{2^w}^{k+2}$ together with $\sqcup, \sqsubseteq$ form a complete lattice whose height is at most $(k+2) \cdot w$. The generator set of vectors for a $\mathbb{Z}_{2^w}$-module $\subseteq \mathbb{Z}_{2^w}^{k+2}$ is generated by at most $k+2$ many vectors [21]. Thus, these generator sets provide an *effective representation* of modules of vectors.

We obtain all valid affine equalities by computing the dual basis for a generator set of vectors. An affine equality relation t over $\mathbb{Z}_{2^w}^{k+2}$ is an equation of the form $t_c + t_0\mathbf{x}_0 + \ldots + t_{k+1}\mathbf{x}_{k+1} = 0$ for $t_i \in \mathbb{Z}_{2^w}$ which is representable by the column vector $(t_c, t_0, \ldots, t_{k+1})^T \in \mathbb{Z}_{2^w}^{k+2}$. Henceforth, the set of all valid affine equality relations is given in form of a set of vectors. A vector $x \in \mathbb{Z}_{2^w}^{k+2}$ satisfies an affine equality $t = (t_c, t_0, \ldots, t_{k+1})^T$ iff its scalar product yields $x \cdot t = 0$. Thus, we obtain a valid affine equality relation $t \in \mathbb{Z}_{2^w}^{k+2}$ for a given module M iff $\forall m \in M : m \cdot t = 0$. We define the function *dual* : $2^{\mathbb{Z}_{2^w}^{k+2}} \rightarrow 2^{\mathbb{Z}_{2^w}^{k+2}}$ to compute the dual basis for a given generator set of vectors. Consider e.g. the generator set $S = \langle (1, 4, 6, 7, 9)^T, (0, 0, 9, 1, 0)^T \rangle$, given the processor registers $\{\mathbf{x}_1, \ldots, \mathbf{x}_4\}$. The dual basis for S is given by the affine equalities $\mathbf{x}_1 = 4$, $9 \cdot \mathbf{x}_3 = \mathbf{x}_2 + 57$, $\mathbf{x}_4 = 9$.

We set up a constraint system, whose least solution specifies the set of program states attainable when reaching a distinguished program point. With the abstraction by generator sets, all affine equalities valid at a program point u are obtained by computing a dual basis for the module of vectors valid at u. The framework for our interprocedural equality relation analysis is the following: we assume that every call node to a function f is connected with the start node s_f of the called function f. Thus, the module for the start node of a function results from the least upper bound of all the call contexts — we take the *call-string approach of length zero* (CSA-0). Encountering an unknown function call, we must assume that any function is called. If there arise infinitely many possible targets for the function call, we lose all information about the register assignment of the caller. However, if there is a finite set of possible targets for a function call, we non-deterministically connect to all the function start points. Thus, at least some information about the global register assignment could be retrieved. For this analysis we assume that conditional branching is generalised to non-deterministic branching. Furthermore, we simplify the effect of *load*-instructions $\llbracket \mathbf{x}_i := M_m[p] \rrbracket$ to non-deterministic assignments $\llbracket \mathbf{x}_i :=? \rrbracket$. The effect of a *store*-instruction $\llbracket M_m[p] := \mathbf{x}_i \rrbracket$ is modelled as a *skip*-statement having no effect on the values of registers. For simplicity, here we only consider memory access instructions applied to words, i.e., $m = 4$: $M_4[p]$. We set up constraint system $R^\sharp$ for CSA-0, whose values are generator sets of vectors from $\mathbb{Z}_{2^w}^{k+2}$:

$$\begin{array}{llll}
[\mathrm{R0}^\sharp]\ R^\sharp(s_{main}) & \sqsupseteq \llbracket \mathbf{x}_{ASP} := \mathbf{x}_1 \rrbracket\ \{\mathbf{e}_0, \ldots, \mathbf{e}_{k+1}\} & \text{entry point of } main \\
[\mathrm{R1}^\sharp]\ R^\sharp(v) & \sqsupseteq \llbracket \mathbf{x}_i :=? \rrbracket R^\sharp(u) & \text{for edge } (u, \mathbf{x}_i :=?, v) \\
[\mathrm{R2}^\sharp]\ R^\sharp(v) & \sqsupseteq \llbracket \mathbf{x}_i := t \rrbracket\ R^\sharp(u) & \text{for edge } (u, \mathbf{x}_i := t, v) \\
[\mathrm{R3}^\sharp]\ R^\sharp(s_f) & \sqsupseteq \llbracket \mathbf{x}_{ASP} := \mathbf{x}_1 \rrbracket\ R^\sharp(u) & \text{for edge } (u, f(), \_) \\
[\mathrm{R4}^\sharp]\ R^\sharp(v) & \sqsupseteq combine(R^\sharp(u),\ R^\sharp(r_f)) & \text{for edge } (u, f(), v)
\end{array}$$

$\mathbf{e}_0, \ldots \mathbf{e}_{k+1}$ denote the unit vectors. We assume that program execution always starts with a call to the specific function `main`. The first constraint $[\mathrm{R0}^\sharp]$ expresses that at the entry point of `main`, we start with the top element of our lattice, which demonstrates that nothing is known about the register assignment. To consistently describe memory accesses with respect to the stack pointer and verify the absence of dynamic stack modifications, we transfer the initial value of the stack pointer $\mathbf{x}_1$ to the artificial stack pointer $\mathbf{x}_{ASP}$ at the start node of every function. This allows for a consistent indication of the beginning of a new stack frame throughout the whole body of a function.

Every transition to another program state induces a transformation of the module of vectors, according to the edge annotation. The effect of a linear assignment ($[\mathrm{R2}^\sharp]$) can be realised by multiplication with a transition matrix [21] — whereas the effect of a non-deterministic assignment $\llbracket \mathbf{x}_i :=? \rrbracket$ ($[\mathrm{R1}^\sharp]$) consists in the union of assigning every possible constant value to $\mathbf{x}_i$ which causes a loss of information about the values of $\mathbf{x}_i$. Technically, this can be implemented by adding the unit vector $\mathbf{e}_i$ as a new generator to the given module of vectors.

Finally, constraints $[\mathrm{R3}^\sharp]$ and $[\mathrm{R4}^\sharp]$ describe the handling of a call to function f. For a function call $f()$ the assignment of the global registers within the caller has to be propagated to the start node of the called function s_f. We assume that the non-volatile registers within the callee are initialised properly and thus, we spare to set these registers to unknown values. This is specified by constraint $[\mathrm{R3}^\sharp]$. In order to embed the effect of a function call into the caller, we *combine* the module of vectors valid before the function was called $R^\sharp(u)$ with the procedural effect $R^\sharp(r_f)$. Within CSA-0, the effect of f is given by the module of vectors valid at its return point r_f. We define $M|_G$ as the restriction of the vectors of the module M to those vectors which only consist of components from G, i.e., for all vectors of M, we omit all those components $\notin G$. Henceforth, G is the set of global processor registers, for PPC i.e. $\{\mathbf{x}_{14}, \ldots, \mathbf{x}_{31}\}$, whereas L is the set of local processor registers, for PPC i.e. $\{\mathbf{x}_1, \ldots, \mathbf{x}_{13}\}$. This means that the module $M|_G$ induces only relations between global registers. For a function call we consider the module of the caller $R^\sharp(u)|_{L\cup\mathbf{x}_0\cup\mathbf{x}_{ASP}}$ inducing only local register relations and the module of the callee $R^\sharp(r_f)|_{G\cup\mathbf{x}_0}$ inducing only global register relations. Then, the tensor product $\otimes$ of the two modules is computed. Geometrically the tensor product corresponds to the Cartesian product of all the points, represented by the two modules. Thus, the function $combine : 2^{\mathbb{Z}_{2^w}^{k+2}} \times 2^{\mathbb{Z}_{2^w}^{k+2}} \to 2^{\mathbb{Z}_{2^w}^{k+2}}$ is defined as

$$combine(R^\sharp(u), R^\sharp(r_f)) : R^\sharp(u)|_{L\cup\mathbf{x}_0\cup\mathbf{x}_{ASP}} \otimes R^\sharp(r_f)|_{G\cup\mathbf{x}_0}$$

Consider the following example:

Example 7. Here, we assume the register set $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_4\}$, where $\mathbf{x}_1, \mathbf{x}_2$ are global registers, $\mathbf{x}_3, \mathbf{x}_4$ have local scope and $\mathbf{x}_0$ denotes the extra $0-$th component. We consider the two generator sets of vectors $M_1 = \langle(1,1,6,1,2)^T, (0,0,1,0,1)^T\rangle$ and $M_2 = \langle(1,0,2,4,1)^T, (0,1,2,1,0)^T\rangle$. The dual basis for M_1 is $\mathbf{x}_1 = 1,\ \mathbf{x}_3 = 1,\ \mathbf{x}_2 = \mathbf{x}_4{+}4$, the affine equalities for M_2 are given by $\mathbf{x}_2 = \mathbf{x}_1 + 2,\ \mathbf{x}_3 = \mathbf{x}_1 + 4,\ \mathbf{x}_4 = 1$. Now, we compute $M_1|_{G\cup\mathbf{x}_0} \otimes M_2|_{L\cup\mathbf{x}_0}$ with

$$M_1|_{G\cup\mathbf{x}_0} = \left\{ \begin{array}{l} \mathbf{x}_0 \to \\ \mathbf{x}_1 \to \\ \mathbf{x}_2 \to \end{array} \begin{pmatrix} 1 \\ 1 \\ 6 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}, M_2|_{L\cup\mathbf{x}_0} = \left\{ \begin{array}{l} \mathbf{x}_0 \to \\ \mathbf{x}_3 \to \\ \mathbf{x}_4 \to \end{array} \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}$$

Geometrically spoken, for the tensor product, we compute the Cartesian product of all the points of the two modules and adopt all the rays as it stands to the result module. Geometrically, all those vectors whose 0-th component is 0 are interpreted as rays. The tensor product yields the module $\langle(1,1,6,4,1)^T, (0,0,1,0,0)^T, (0,0,0,1,0)^T\rangle$, which corresponds to the affine equalities $\mathbf{x}_4 = 1,\ \mathbf{x}_1 = 1$. □

Here, we assume that the assembly code adheres to the calling conventions from the processor ABI or a preceding analysis verified that the local registers were saved and additionally restored properly. After applying the function effect, we obtain a module of vectors implying the global register assignment of the callee and the local register assignment of the caller.

Summing up, we provided a description of the transfer functions of constraint system $R^\sharp$, which allows us to determine all valid affine register equalities for very program point. All right-hand sides of the inequations in constraint system $R^\sharp$ denote monotonic functions and thus $R^\sharp$ has a unique least solution. The termination of the fixpoint iteration over the constraint system $R^\sharp$ can be guaranteed, because every descending chain of submodules of vectors in the finite domain $\mathbb{Z}_{2^w}^{k+2}$ will finally terminate and because of the monotonicity of the transfer functions.

For establishing a complexity bound for the equality relation analysis, we assume a program size of n program points, $k+2$ processor registers and a bit width of w. Within the fixpoint iteration at each of the n program points, we can extend the existing module at most $w \cdot (k+2)$ times with costs of $\mathcal{O}(w \cdot (k+2)^2)$ every time. In order to obtain all valid affine equality relations for one distinguished program point, we have to solve a system of linear equations over $\mathbb{Z}_{2^w}^k$, which can be performed in $\mathcal{O}(log(w) \cdot (k+2)^3)$ [21]. Summarising, we arrive at a total complexity of $\mathcal{O}(n \cdot w^2 \cdot (k+2)^3)$.

Additionally, our approach allows to verify that there occurs no dynamic modification of the stack frame of the caller. For this purpose we investigate if there was a dynamic modification of the stack frame of the callee, which could have influenced the stack frame of the caller, as e.g. example 2 illustrates. Since we assume that the safety-critical code we analyse adheres to the characteristics specified in section 2, our implementation issues a warning if the stack level of the caller could have been changed by a function call. In order to detect a modification of the stack frame of the caller after a function call, we define the auxiliary function $\delta : \mathbb{Z}_{2^w}^{k+2} \times 2^{\mathbb{Z}_{2^w}^{k+2}} \to (\mathbb{Z}_{2^w})_\perp$. δ checks if the global invariant t holds for a given set of affine equalities S.

Algorithm 1 $\delta(t, S)$

```
if exists $c \in \mathbb{Z}_{2^w}$ $\forall x \in S : \sum_i t_i x_i = c$
    return $c$;
else
    return $\perp$;
```

After a function call $f()$, we check if the difference between the stack pointer and the artificial stack pointer $\mathbf{x}_1 - \mathbf{x}_{ASP}$ (SP-ASP relation) is equal for all affine equalities from *dual*$(R^\sharp(r_f))$. If this invariant holds, we obtain a constant c describing the SP-ASP difference – otherwise the value $\perp$ is returned. If the constant is equal to zero, the stack frame established by the callee was successfully popped from the stack and has no influence on the stack pointer of the caller. Otherwise we issue a warning that the stack pointer of the caller has to be increased by this offset. However, if the invariant does not hold ($\perp$ is returned), it was not possible to restore the SP-ASP relation of the callee. Again, we issue a warning which states that we would have to discard all information about the stack pointer of the caller after the function call in order to provide save results.

Now, we describe a *may-analysis* which classifies memory locations as possible local or global variables using the previous inferred affine register equalities.

Identifying Local Variables

Since only memory access instructions ($\mathbf{x}_i := M_m[p]$ res. $M_m[p] := \mathbf{x}_i$) access and if so possibly modify memory [28], we are interested for every of these instructions which address in memory is actually accessed. Memory accesses may originate from local or global variables or arrays. Since we cannot distinguish accesses to variables from constant accesses to arrays, we present a *may analysis* inferring candidates for local and global variables. A memory access, e.g., $M_m[\mathbf{x}_i]$ *possibly* accesses a local variable iff there exists a constant $c \in \mathbb{Z}_{2^w}$, such that the relation $\mathbf{x}_i - \mathbf{x}_{ASP} = c$ holds. This can be achieved by verifying that the previously inferred valid affine equality relations imply this relation. For an edge (u, s, v) where s contains a memory access expression $M_m[p]$ we check if $\delta(p - \mathbf{x}_{ASP}, dual(R^\sharp(u)))$ yields a constant $c \in \mathbb{Z}_{2^w}$. In case of a global variable we have to prove that there exists a constant $c \in \mathbb{Z}_{2^w}$, such that the relation $p = c$ holds with respect to all valid affine equalities. The function $memacc$ accumulates candidates for local and global variables in the set $\mathbf{L}$ of local variables and the set $\mathbf{G}$ of global variables. For every memory access $M_m[p]$ information is provided whether the stack or the global data region is accessed and if so exactly at which address.

Algorithm 2 $memacc(R^\sharp(u), \mathbf{L}, \mathbf{G})$ with edge $(u, \mathbf{x}_i := M_m[p]$ res. $M_m[p] := \mathbf{x}_i, v)$

```
if $(\delta(p, dual(R^\sharp(u))) = c)$
    $\mathbf{G} \leftarrow \mathbf{G} \cup \{\mathbf{x}_{-c}\}$     // global variable at address $c$
else if $(\delta(p - \mathbf{x}_{ASP}, dual(R^\sharp(u))) = c)$
    $\mathbf{L} \leftarrow \mathbf{L} \cup \{\mathbf{x}_{k+1+c}\}$     // local variable at address $\mathbf{x}_{ASP} + c$
else    // unknown memory access
```

If a memory location is classified as local variable with address $\mathbf{x}_{ASP} + c$, the set of local variables $\mathbf{L}$ is extended by this memory location. In order to provide a unique name for this local variable with address $\mathbf{x}_{ASP} + c$, we denote it by $\mathbf{x}_{k+1+c}$. When a global variable with address c was identified, the set $\mathbf{G}$ is extended by a new global variable, which is uniquely addressed by $\mathbf{x}_{-c}$, henceforth.

Consider the following example whose C-code conforms to the characteristics from section 2.

Example 8. variable identification

```
int a[4]={1,2,3,4};
int main(){
  int i,j;
  for(i=0;i<4;i++){
    j += a[i];
  }
  return j;
}
```

```
01: stwu   r1,-32(r1)
//for(i=0;i<4;i++){
02: li     r0,0
03: stw    r0,12(r1)
04: b      0x17
//     j += a[i];
05: lwz    r0,12(r1)
06: lis    r9,10
07: addi   r9,r9,6556
08: rlwinm r0,r0,2,0,29
09: add    r9,r0,r9
```

```
10: lwz    r9,0(r9)
11: lwz    r0,8(r1)
12: add    r0,r0,r9
13: stw    r0,8(r1)
14: lwz    r9,12(r1)
15: addi   r0,r9,1
16: stw    r0,12(r1)
17: lwz    r0,12(r1)
18: cmpwi  cr7,r0,3
19: ble    cr7,0x05
.
.
```

In this example only local variables and a global array are addressed. Our technique of identifying local variables correctly identifies all occurring variables, i.e., the local variables $\_i$ at address $r1 - 20$ (complying with the C-variable i), $\_j$ at address $r1 - 24$ (complying with the C-variable j). Instruction 10 suggests that the global data section is accessed. For the first iteration of the loop our analysis yields that the global address $0xa199c$ is addressed. In order to provide information about the structure of variable $\_a$ an interval analysis is required, taking the accesses to the single array cells into account. This is elaborated in section 4.

Summarising, this analysis provides *may*-information, classifying every memory access as stack related, global variable, or if none of these two cases applies, as an unknown memory access. Furthermore, we obtain a normalised representation of address expressions through our affine equality relations. That means, e.g., if a local variable is addressed the address expression can be reduced to a constant stack pointer offset.

Escaping Variables So far, we have categorised memory accesses as potential local or global variables or unknown memory accesses. We have to take into account that the addresses of the inferred candidates for local variables may lie within the address bounds of an array. Thus, if any element of the array is written, we have to invalidate the values of the corresponding candidates. Consider the following example where the address of the local variable j is passed as parameter to function f.

Example 9. escaping variables

```
void f(int *k){     //main:                 //f:
  k++;              01:  .                  11:  stwu   r1,-16(r1)
  *k=5;             02:  .                  12:  stw    r3,8(r1)
}                   03:  li     r0,0        13:  lwz    r9,8(r1)
                    04:  stw    r0,8(r1)    14:  addi   r0,r9,4
int main(){         05:  addi   r0,r1,12    15:  stw    r0,8(r1)
  int i,j;          06:  mr     r3,r0       16:  lwz    r9,8(r1)
  i=0;              07:  bl     0x11        17:  li     r0,5
  f(&j);            08:  lwz    r0,8(r1)    18:  stw    r0,0(r9)
  return i;         09:  .                  19:  addi   r1,r1,16
}                   10:  .                  20:  blr
```

Function f modifies the address the formal parameter k points to (cf. instruction 14). Thus, k then points to $main$'s local variable i instead of j. Subsequently the value of i is modified by the assignment to k (cf. instruction 18), though, i was not directly passed to function f. Consequently, for this example the only safe assumption is to discard all information about all the potential local variables of function $main$ after the call to f. Because our analysis does not ensure that the callee may only modify a small range of stack cells within the caller.

Since our analysis of safety-critical assembly code assumes that there is neither pointers nor address arithmetic, the situation of example 9 should not occur in the programs we consider. Nevertheless, we want to verify if a present assembly code adheres to this coding conventions.

Thus, it is our goal to indicate whether the address of a local variable may *escape* or not, i.e., whether its address is made accessible for code outside of the current function. In example program 9, instruction 06 causes that the address of local variable j is written to the parameter passing register $r3$. Within the callee, instruction 14 performs an access to the stack frame of the caller and modifies its contents as instruction 18 illustrates. In order to detect such modifications of the stack frame of the caller, we want to determine within the caller (for example 9 i.e. function `main`) whether the value of the operand of any *store*-instruction $M_m[p] := \mathbf{x}_i$, i.e. $\mathbf{x}_i$ is dependent on the stack pointer (i.e. contains a stack address). As the first actual parameters are exchanged with the caller via registers (in PPC, i.e., registers $\mathbf{x}_3, \ldots, \mathbf{x}_{10}$ [28]), we must also check if a stack address is saved in one of these registers directly before a function is called.

Example 10. criteria for escaping stack addresses

```
int g;
int main(){
  int e;
  f(&e);
  g=&e;
  return &e;
}
```

The address of a local variable is marked as *escaping* the current function if it ...

- ... is passed as a parameter,
- ... is written into a global variable,
- ... is passed as return value of the current function, or
- ... is written into memory.

When storing the address of a local variable and one of these characteristics applies, we have to treat memory as tainted. Escaping variables result in a loss of all information about these variables whenever an unknown memory location is written or a function was called. Unknown memory location means that our variable identification analysis was not able to classify this memory location as stack address or global address. Now, consider the following situation: We assume that memory is tainted and encounter a *load*-instruction. Then, we have to discard all information about all the potential local variables, because we do not know whether other stack addresses or their values may be modified by instructions following this *load*-instruction. Within our implementation we issue a warning if a stack address may escape, since we assume that there is no pointer arithmetic and only global arrays are considered. This states that the present assembly code may not conform the requested coding guidelines. In order to provide information whether the address of a local variable could escape or not, we examine *store*-instructions. Encountering an edge $(u, M_m[p] := \mathbf{x}_i, v)$ in the context that the set of affine equalities $\mathit{dual}(R^\sharp(v))$ is valid, we are interested if $\mathbf{x}_i$ contains the address of a local variable, i.e., may be dependent of the artificial stack pointer. In order to discover a stack pointer dependency, we check the invariant that the difference between the memory access register $\mathbf{x}_i$ and the artificial stack pointer yields a constant. If this invariant holds $\delta(\mathbf{x}_i - \mathbf{x}_{ASP}, \mathit{dual}(R^\sharp(v))) \neq \bot$, the value of $\mathbf{x}_i$ definitively shows a dependency of the stack pointer, and thus taints memory. Additionally, we have to check whether $\delta(\mathbf{x}_i, \mathit{dual}(R^\sharp(v)))$ yields a constant. If this is the case, a global value is saved. Otherwise, it is not clear in any case whether a local variable is addressed or not.
Concluding, the approach of identifying escaping stack addresses is used to verify that a given assembly code is free of pointers and address arithmetic.

4 Extended Interval Analysis

When considering the iterations of loops, the array cells within the loop are often accessed relative to a loop counter variable, which mostly turns out to be a local variable. In order to infer bounds for the accessed memory cells, the bounds of the loop counter are required.

Extended Interval Analysis
To provide safe lower and upper bounds for the values of processor registers, an interval analysis is performed. Here, every state maps processor registers and local res. global variables to intervals of possible values from $\mathbb{Z}_{2^w}$. A detailed description of an interval analysis on assembly code can be found in [12], where the case of overflows in processor arithmetic over $\mathbb{Z}_{2^w}$ is also addressed.
Here, we consider the extended interval domain $(\mathbf{X} \cup \mathbf{L} \cup \mathbf{G} \to \mathbf{I})_\bot$ with $\mathbf{I}$:

$$\mathbf{I} = \{(s,[l,u]) \mid l \in \mathbb{Z}_{2^w} \cup \{-\infty\}, u \in \mathbb{Z}_{2^w} \cup \{+\infty\}, l \leq u, s \in \{0,1\}\}$$

The flag s is intended to denote the stack pointer dependency, which is definitively not given if $s = 0$ — in case of an address expression, i.e., that the global data section is addressed. If $s = 1$ the corresponding interval must be considered relative to the stack pointer — for an address expression, i.e., local variables on the stack are addressed. For this interval analysis, the register set $\mathbf{X}$ only consists of processor registers $\{\mathbf{x}_1, \ldots, \mathbf{x}_{31}\}$. For every procedure start, the stack pointer $\mathbf{x}_1$ has to be initialised with the interval $(1,[0,0])$. Since the value of $\mathbf{x}_{ASP}$ is constantly equal to interval $(1,[0,0])$ we omit tracking its value within the extended interval analysis.
The least element is denoted by $\bot$, the greatest element by $\top$. Two elements from $\mathbf{I}_\bot$ are only comparable to each other if their flag s has the same value. Then, the ordering $\sqsubseteq$ is defined as $(s_1,[l_1,u_1]) \sqsubseteq (s_2,[l_2,u_2])$ iff $s_1 = s_2 \wedge l_1 \leq l_2 \wedge u_1 \leq u_2$. Otherwise they are considered incomparable. Addition res. subtraction of two extended intervals $(s_1,[l_1,u_1]),(s_2,[l_2,u_2])$ is realised by adding res. subtracting the two intervals according to the interval semantic and the result flag s is computed by:

$+$	$s_2 = 1$	$s_2 = 0$
$s_1 = 1$	$\top$	1
$s_1 = 0$	1	0

$-$	$s_2 = 1$	$s_2 = 0$
$s_1 = 1$	0	1
$s_1 = 0$	1	0

$\cdot$	$s_2 = 0$
$s_1 = 1$	$\top$
$s_1 = 0$	0

Multiplication with a constant $\in \mathbb{Z}$ leaves the flag unchanged iff $s_1 \neq 1$, otherwise it leads to $\top$. The least upper bound of two extended intervals $(s_1,[l_1,u_1]) \sqcup_I (s_2,[l_2,u_2])$ is defined as $(s_1,[l_1 \sqcap l_2, u_1 \sqcup u_2])$ if $s_1 = s_2$ and $\top$ otherwise.
To conclude, this slightly modified interval analysis is the basis for inferring bounds both for registers and memory locations. We arrive at the following constraint system I:

[I0] $I(s_{main}) \sqsupseteq \top$
[I1] $I(v) \quad \sqsupseteq I(u) \oplus \{\mathbf{x}_i \mapsto \top\}$ $\quad (u, \mathbf{x}_i :=?, v)$
[I2] $I(v) \quad \sqsupseteq I(u) \oplus \{\mathbf{x}_i \mapsto [\![t]\!]^\sharp_{\mathrm{I}} I(u)\}$ $\quad (u, \mathbf{x}_i := t, v)$
[I3] $I(s_f) \quad \sqsupseteq I(u) \oplus \{\mathbf{x}_1 \mapsto (1,[0,0])\}$ $\quad (u, f(), \_)s_f$ entry point of f
[I4] $I(v) \quad \sqsupseteq combine_I(I(u),\ I(r_f))$ $\quad (u, f(), v)$
[I5] $I(v) \quad \sqsupseteq I(u) \oplus \{\mathbf{x}_i \mapsto [\![\mathbf{x}_i \,\square\, \mathbf{x}_j]\!]^\sharp_{\mathrm{I}} I(u)\}$ $(u, \mathbf{x}_i \,\square\, \mathbf{x}_j, v)$with $\square \in \{\geq, >, \leq, <\}$
[I6] $I(v) \quad \sqsupseteq I(u) \oplus \{\mathbf{x}_i \mapsto [\![\mathbf{x}_i \,\square\, c]\!]^\sharp_{\mathrm{I}} I(u)\}$ $(u, \mathbf{x}_i \,\square\, c, v)$with $\square \in \{\geq, >, \leq, <\}$

Program execution starts with the entry point of main where all registers and memory locations are set to unknown intervals [I0]. In case of a non-deterministic assignment to register $\mathbf{x}_i$ its interval is set to the top element of our lattice losing all information about this register, as constraint [I1] states. For a linear assignment [I2], the linear term on the right-hand side is evaluated according to the interval semantic [12] — this is denoted by the semantic brackets $[\![\ ]\!]^\sharp_{\mathrm{I}}$. For a guarded transition the condition is also evaluated according to the extended interval semantic and causes that the interval for the left-hand side variable of the condition is modified. This is stated in constraints [I5] and [I6]. Notice, however, that the value of the right-hand side variable is only modified if its flag is equal to the flag of the evaluated condition. For details about condition handling in the interval domain refer to [12].

Example 11. Consider e.g. condition $(\mathbf{x}_i > \mathbf{x}_j)$, where $\mathbf{x}_i \mapsto (1, [1, 8])$ and $\mathbf{x}_j \mapsto (1, [2, 4])$. After passing the *true*-edge, the interval of $\mathbf{x}_i$ is restricted to $(1, [5, 8])$. For the *false*-edge, we obtain $\mathbf{x}_i \mapsto (1, [1, 4])$. Now, we assume the register assignment $\mathbf{x}_i \mapsto (1, [1, 8])$ and $\mathbf{x}_j \mapsto (0, [2, 4])$ before passing the condition. Then, the interval for $\mathbf{x}_i$ is neither modified along the *true*-edge nor along the *false*-edge, but propagated unchanged along both edges. □

Constraint [I3] specifies the handling of a call to function f. Since we take CSA-0 and assume that the local registers and variables are initialised properly before they are used, the register assignment before a function call is transferred to the start node of the called function. Additionally the stack pointer of the callee is reset to the interval $(1, [0, 0])$. The algorithm $combine_I$ realises embedding the effect of a function call into the caller ([I4]). $combine_I$ works analogously to the algorithm $combine$ over generator sets of vectors from section 3.

$$combine_I(I(u), I(r_f)) : \begin{cases} I(r_f)(\mathbf{x}_i) & \text{with } \mathbf{x}_i \notin \{\mathbf{x}_1, \mathbf{x}_{14}, \ldots, \mathbf{x}_{31}\} \cup \mathbf{L} \\ I(u)(\mathbf{x}_i) & \text{with } \mathbf{x}_i \notin \{\mathbf{x}_2, \ldots, \mathbf{x}_{13}\} \cup \mathbf{G} \end{cases}$$

The values for non-volatile registers and all the local variables within the callee $I(r_f)$ are masked out. Likewise all the values for global registers and all the global variables of the caller $I(u)$. The application of the function effect results in intervals reflecting the global variable/register assignment of the callee and the local variable/register assignment of the caller.

Termination of this fixpoint iteration is ensured because of the monotonicity of the transfer functions and the fact that a register res. memory location can only hold finitely many different values. Efficiency can only be ensured by applying suitable widening operators [8].

Recapitulating, our extended interval analysis computes for every processor register and memory location an approximation of all possible values (constants or offsets with respect to the stack pointer) occurring during runtime. Additionally the extended interval analysis contributes much in inferring more precise results on identifying and handling local and global variables.

Compiler Optimisation Level

In the area of safety-critical real-time applications a compiler optimisation level of zero or even less is often used. The output of a compilation step with optimisation level zero causes that the values of local variables are always freshly loaded from memory. Consider the following *unoptimised assembly code* 12 where two local variables i and j in C code are addressed. When we take a closer look at instruction 04, the value of local variable $\_i$ with address $r1 + 12$ (complying with local variable i in C code) is loaded into register $r0$. Although the *load*-instruction 04 is unnecessary, because register $r0$ already contains the value of local variable $\_i$ at this program point, it is performed regardless of available expressions. The interval analysis holds the interval $[4, \top]$ for register $r0$ at instruction 04. However, the intervals for processor registers and variables are kept separately and the inferred interval for $r0$ is never transferred to local variable $\_i$ before passing instruction 04. Thus, no precise intervals for the local variables $\_i$ and $\_j$ (memory address $r1 + 8$) can be inferred. Applying naive interval analysis on zero-optimised assembly code thus mostly leads to useless results, as almost no array bound can be found.

Example 12. unoptimised assembly code

```
int main(){                         01:     lwz      r0,12(r1)
  int i,j;                          02:     cmpwi    cr7,r0,3
  if(i>3)                           03:     ble      cr7,0x06
    j=i;                            04:     lwz      r0,12(r1)
  return -1;                        05:     stw      r0,8(r1)
}                                   06:     li       r0,-1
```

The intuitive solution for this problem is to *propagate* the changes of intervals for processor registers *back* to the variables they are buffering. Consider instruction 01: this *load*-instruction means that the value of local variable $\_i$ is buffered in register $r0$. Thus, after instruction 01 the value of register $r0$ and variable $\_i$ are equal. We say that register $r0$ *buffers* local variable $\_i$. To yield precise intervals for variables, we have to provide information which register is buffering a variable.

Relating Registers and Memory Locations

In order to provide information about the coherence of memory locations and registers, we need equality relations comprising not only registers but also local and global variables. We obtain these equalities by extending the equality relation analysis from section 3 to include memory locations. Dealing with local and global variables entails modifying the algorithms $memacc$ and $combine$. The algorithm $memacc3$ now works as follows: In case of an access to a familiar memory location, we perform an assignment between the register and the variable corresponding to the memory location (this assignment states an equality between the register and this variable). In case of an unknown write access, we must invalidate all the information about the local variables $\mathbf{L}$ in order to provide save results. An unknown read access causes a non-deterministic assignment to the destination register.

	$(\mathbf{u}, \mathbf{x_i} := \mathbf{M_m}[\mathbf{p}], \mathbf{v})$	$(\mathbf{u}, \mathbf{M_m}[\mathbf{p}] := \mathbf{x_i}, \mathbf{v})$
$\delta(p, \mathit{dual}(R^\sharp(u))) = c$	$[\![\mathbf{x}_i := \mathbf{x}_{-c}]\!]\ R^\sharp(u)$	$[\![\mathbf{x}_{-c} := \mathbf{x}_i]\!]\ R^\sharp(u)$
$\delta(-\mathbf{x}_{ASP} + p, \mathit{dual}(R^\sharp(u))) = c$	$[\![\mathbf{x}_i := \mathbf{x}_{k+1+c}]\!]\ R^\sharp(u)$	$[\![\mathbf{x}_{k+1+c} := \mathbf{x}_i]\!]\ R^\sharp(u)$
otherwise	$[\![\mathbf{x}_i :=?]\!]\ R^\sharp(u)$	$\forall \mathbf{l} \in \mathbf{L} :\ [\![\mathbf{l} :=?]\!]\ R^\sharp(u)$

Table 1. Side effects of algorithm $memacc$

Another modification in presence of local res. global variables in affine equality relation analysis affects the algorithm $combine$ for applying a function effect. The local variables of the caller and the callee belong to different scopes. Therefore, relations concerning local variables of the callee must not modify relations concerning local variables of the caller. However, relations concerning global variables of the callee become the new relations concerning global variables of the caller. Thus, the function $combine$ is now defined as

$$combine(R^\sharp(u), R^\sharp(r_f)) : R^\sharp(r_f)|_{L \cup \mathbf{x}_0 \cup \mathbf{x}_{ASP} \cup \mathbf{L}} \otimes R^\sharp(u)|_{G \cup \mathbf{x}_0 \cup \mathbf{G}}$$

In summary, the extended equality relation analysis states equalities between processor registers and memory locations.

Analysis Refinement

In order to achieve more precise results, we refine our analysis framework by interweaving affine equality relation analysis and interval analysis. Therefore, we consider the *reduced product* [7] of the extended equality relation domain and the extended interval domain. In both domains besides the processor registers $\mathbf{X}$, we additionally take into account memory locations given by $\mathbf{L}$ and $\mathbf{G}$. Precision gains through the reduced product approach result form the mutually improving collaboration of the two analyses.

The algorithm $reduce$ transfers information from the equality relation domain R to the interval domain I and vice versa. Thus, the effect of a statement s results in:

$$[\![s]\!](R, I) = \text{let } R' = [\![s]\!]^\sharp R,\ I' = [\![s]\!]^\sharp_{\mathrm{I}} I \ \text{ in } reduce(R', I', s)$$

The algorithm $reduce$ works as follows. At every program point u, first, it is checked whether $\mathbf{x}_i$ buffers a memory location, i.e., $\mathbf{x}_i = \mathbf{ml}$ with $\mathbf{ml} \in \mathbf{L} \cup \mathbf{G}$ is implied by all affine equalities valid at u. If this is the case the interval for memory location $\mathbf{ml}$ is refined by the interval inferred for $\mathbf{x}_i$, i.e. $I' = I \oplus \{\mathbf{ml} \mapsto I(v)(\mathbf{x}_i)\}$.

In case of a *store*-instruction $(u, M[p] := \mathbf{x}_i, v)$, where the target address is unknown, the information from the interval domain affects the information of the equality domain. Since the interval analysis may provide some information about former unknown memory accesses, we modify the effect of an unknown *store*-instruction from table 1. That means that we spare to set all the inferred local variables to an unknown value. Then, if we know that the stack section and not the global data section is affected, i.e., the address expression p evaluates to an interval $(1, [l, u])$ all those local variables, whose addresses lie in the range $(1, [l, u])$, have to be set to unknown values in the equality domain $\{[\![\mathbf{x}_{k+1+l} :=?]\!] \dots [\![\mathbf{x}_{k+1+u} :=?]\!]\}R^\sharp(v)$ as well as within the interval domain $I(v) \oplus \{\forall\, \mathbf{l} \in \mathbf{L} :\ \mathbf{l} \mapsto \top\}$. If we can exclude that the stack is addressed, then all those global variables, whose addresses lie in the specified range, have to be set to unknown values. Otherwise, information both for local and global variables gets lost.

In practise, the mechanism of back propagating intervals to memory locations allows to additionally infer an interval of possible values for each local res. global variable. We can draw conclusions from these intervals about the destination region of basic stack pointer-indirect memory accesses. Thus, we are able to invalidate register values more selective when dealing with non-exact memory accesses. Reconsidering example 12 for zero-optimised assembly code, *compare*-instruction 02 yields that register $r0$ implies the extended interval $(0, [4, \top])$ at instruction 04. With the help of the analysis refinement, this interval is propagated to memory location *_j* after instruction 05 is passed.

Consider the following example, which demonstrates the impact of the reduced product approach: There is a constant memory access $a[2]$, which is classified as a local variable within our equality relation analysis.

Example 13. array access

```
int a[5],i,j;
a[2] = 1;
j = 5;
for(i=0; i<5;i++)
      a[i] = i;
```

Then, we identified the write access to the whole address range of array a. Since the address of the local variable $a[2]$ lies in the range of the array access in line 5, we have to set its value unknown — both in the interval domain and the affine equality relation domain. For subsequent write accesses to $a[2]$ the corresponding local variable has to be considered again. Furthermore, the values for the memory locations _i and _j stay untouched.

Dependent on its implementation, the interval analysis could yield very coarse results for the address range of the array. For example 13, we could obtain e.g. these results:

- variable i in C code is identified as memory location _i at address $\mathbf{x}_{ASP} + 4$
- variable j in C code is identified as memory location _j at address $\mathbf{x}_{ASP} + 12$
- for _i the interval $(1, [12, 32])$ was inferred
- a in C code is identified as memory location _a at address $\mathbf{x}_{ASP} + [12, 32]$

When we have analysed memory access instruction $a[i] = i$, i.e., inferred a new interval for the value of variable _a, we must discard all information for the local variable _j in order to provide a safe analysis – since the address of _j lies in the approximation of the address range for _a. Thus, in case of a write access we must discard information about all those local variables whose addresses lie in the specified address range of the write access in order to yield correct results again. It has to be mentioned that the interval analysis we used [12] yields precise bounds for the loop counter variable in example 13.

Concluding we present an example 14, which iteratively accesses the single elements of a local array, and demonstrate the results our refined analysis produces.

Example 14. array access

```
                          01:  li     r0, 0       09:  li     r0,3
                          02:  stw    r0,8(r1)    10:  stw    r0,0(r9)
                          03:  b      0x14        11:  lwz    r9,8(r1)
int i,a[100];                                     12:  addi   r0,r9,1
for(i=0;i<100;i++)        04:  lwz    r0,8(r1)    13:  stw    r0,8(r1)
   a[i] = 3;              05:  mulli  r9,r0,4
                          06:  addi   r0,r1,8     14:  lwz    r0,8(r1)
                          07:  add    r9,r9,r0    15:  cmpwi  cr7,r0,99
                          08:  addi   r9,r9,4     16:  ble    cr7,0x04
```

The result of the preceding variable identification analysis yields that the local variable i in C code is addressed by $M_4[r1 + 8]$ in assembly. Our equality relation analysis introduces the local variable _i for memory location $M_4[r1+8]$. After having analysed instruction 14, the linear equality relation $r0 =$ _i holds. Instructions 04 through 08 describe the computation of the address of $a[i]$ which is stored in register $r9$. This results in accessing memory at position $r1+8+4\cdot$_$i+4$. Relation $r9 = r1+8+4\cdot$_$i+4$ after passing instruction 08 displays the information that register $r9$ is depending on the value of the stack pointer $r1$ and on the value of the local variable _i. In order to dissolve this address properly the value of variable _i has to be known. The interval analysis [12] provides us with the extended interval $(0, [0, 99])$ for _i at this program point, since the analysis refinement causes that this interval is transferred from register $r0$ to local variable $\_i$ when passing instruction 16. Thus, within the extended interval analysis we obtain the interval with stack pointer dependency $(1, [12, 408])$ for register $r9$. To identify that the single elements of an integer array are accessed, the memory access expression $r1+12+4\cdot\_i$ has to be checked for its alignment, given by the corresponding memory access instruction. Checking the alignment, as described in the next section, refines the interval $(1, [12, 408])$ to the set of stack pointer offsets $\{12, 16, 20, \ldots, 408\}$.

5 Alignment Analysis

As stated in the PPC-ABI [28], the alignment of memory accesses has a crucial impact on their performance and thus leads to more accurate computations of the WCET [2]. For this purpose we are interested in as precise alignment properties as possible. In WCET computation, alignment properties play a role in two different aspects: *memory operand alignment* and *burst access*. First, we describe how memory operand alignment can be checked with the help of the inferred affine register equality relations.

Memory operand alignment As described in [15], according to the length of the operand every operand of a single-register memory access instruction has a natural alignment boundary. This means that the address of an aligned operand should be an integral multiple of its length. When accessing memory operands, the best performance can only be guaranteed if alignment on their natural boundaries is given – otherwise one may meet performance degradation. This alignment property means that an access to an operand of size m bytes at memory address a is aligned iff $a \mod m \equiv 0$. From modular arithmetic, we can use the fact that the validity of an equality relation modulo 2^m implies that the equality relation is as well valid for moduli $2^1 \ldots 2^{m-1}$. It is not necessary to perform additional fixpoint analyses for inferring equality relations valid modulo powers of two less than 2^{m-1}. A linear equality relation $p = 0$ is valid modulo 2^m for $m < w$ iff $2^{w-m} \cdot p = 0$ is valid modulo 2^w.

Hence, for every memory access instruction $(u, \mathbf{x}_i := M_m[p]$ res. $M_m[p] := \mathbf{x}_i, v)$ we want to diagnose whether memory accesses are aligned or not. In this process, we have to check for every vector x from the set of affine equalities $\mathit{dual}(R^\sharp(v))$ whether the equality relation $2^{32-m} \cdot p(x) = 0$ is valid.

In the PPC architecture [28], the stack pointer $\mathbf{x}_1$ is initially $16-$byte aligned, i.e., $\mathbf{x}_1 \mod 16 \equiv 0$. Thus, within the affine equality relation analysis, we add the initial precondition $\mathbf{x}_{ASP} = 2^4 \mathbf{x}_{ASP}$. This initial alignment assumption of the stack pointer yields the following modified constraint $[\mathrm{R0}^\sharp]$ for the start point of `main`.

$$[\mathrm{R0}^\sharp] \; R^\sharp(s_{main}) \sqsupseteq [\![\mathbf{x}_{ASP} := 2^4\mathbf{x}_{ASP};\, \mathbf{x}_1 := \mathbf{x}_{ASP}]\!] \, \langle \mathbf{e}_0, \ldots, \mathbf{e}_{k+1} \rangle$$

For every called function the 16-byte alignment property for the stack pointer is preserved, according to the calling conventions of the processor ABI, even if a dynamic stack modification as, e.g., via `alloca` has taken place. In order to verify if a given assembly code adheres to the coding conventions of the processor ABI, we perform additional checks. Within the alignment analysis we check whether the alignment of the stack pointer $\mathbf{x}_1$ is preserved whenever a write access on the value of the stack pointer is performed or a function is called. We only emanate that at the entry point of function `main` the stack pointer is 16-byte aligned.

Examine the *store*-instruction at address 10 from example 9, the affine register relations $\{-16-\mathbf{x}_1+\mathbf{x}_9-4\cdot\mathbf{x}_0=0,\ -48-\mathbf{x}_1+\mathbf{x}_{ASP}=0,\ 2^4\mathbf{x}_{ASP}=0\}$ hold $\mod 2^{32}$. For the memory access at register $\mathbf{x}_9$, the relation $\mathbf{x}_9=\mathbf{x}_{ASP}+4\cdot\mathbf{x}_0-32$ must be checked for $4-$byte alignment, which means whether this relation $\mod 2^2$ evaluates to zero or not. This is equivalent to a multiplication of this relation with 2^{30}. Thus, we obtain: $(\mathbf{x}_{ASP}+4\cdot\mathbf{x}_0-32) \mod 4 \equiv (\mathbf{x}_{ASP}+0-0) \mod 4$. Since at this program point the artificial stack pointer is 16-byte aligned (as the relation $2^4\mathbf{x}_{ASP} \mod 2^{32} \equiv 0$ states), this memory access is $4-$byte aligned. Consider the following example where the single components of the structure `struct x` are accessed within a loop.

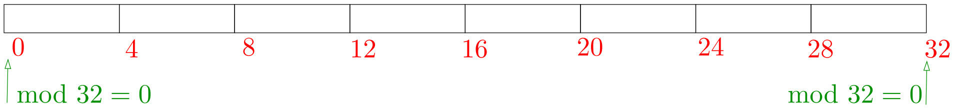
Example 15. handling complex data structures

```
struct x{
  int a;
  int b;               //ar[i].a=1;           //ar[i].b=2;
};                     10:  lwz   r0,8(r1)    17:  lwz   r0,8(r1)
int main(){            11:  mulli r9,r0,8     18:  mulli r9,r0,8
  struct x ar[4];      12:  addi  r0,r1,8     19:  addi  r0,r1,8
  int i;               13:  add   r9,r9,r0    20:  add   r9,r9,r0
  for(i =0;i<4;i++){   14:  addi  r9,r9,4     21:  addi  r9,r9,8
    ar[i].a=1;         15:  li    r0,1        22:  li    r0,2
    ar[i].b=2;         16:  stw   r0,0(r9)    23:  stw   r0,0(r9)
  }
}
```

The *store*-instruction at 16 yields that memory is accessed at position $8\cdot\_i+r1+12$ — at instruction 23 memory is accessed at position $8\cdot\_i+r1+16$. $\_i$ denotes the local variable at memory address $r1+8$. The interval analysis has inferred the interval $[0,3]$ for variable $\_i$. When evaluating the memory access expressions and factoring into the alignment factor (of 4 bytes) denoted by the *store*-instruction for instructions 16 ($r1+[12,44]$) and 8 ($r1+[16,48]$), we detect that their memory cells overlap. In order to distinguish the single components of a complex data structure, we try to reconstruct the set-up of this data structure. For this purpose we determine the greatest alignment factor m with respect to which one of the address expressions p_i is aligned to, i.e., the greatest m such that $deg(2^m\cdot p_i)<1$. In case of our example, we obtain an alignment factor of 8. Thus, at instruction 16 memory is accessed in steps of 8 bytes with an offset of 4 (given as the rest of the modulo operation $p_i \mod 2^m$) with respect to the accessed address space, i.e., $r1+\{12,20,28,36,44\}$. For instruction 23 memory is accessed in steps of 8 bytes with an offset of 0 with respect to the accessed address space, i.e., $r1+\{16,24,32,40,48\}$. This information suggests a data structure of length 8 bytes, consisting of two components. The first component is accessed with offset 0, whereas its second component starts at offset 4.

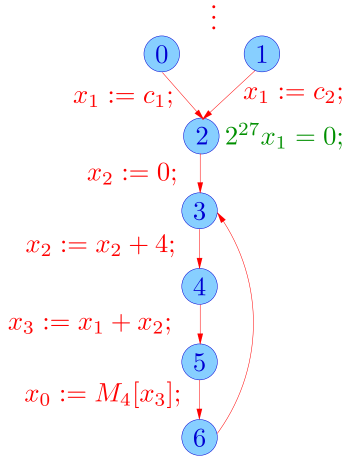
Summarising, for every memory access we use the inferred information to check the valid modulo 2^{32}-relations for operand alignment.

Burst access The second form of alignment concerns the cache. In PPC each cache line contains 32 bytes. Such a cache line is, e.g., loaded as continuous bundle of 8 small 4-byte data blocks into the cache.



In case of a memory access, if data is transferred from consecutive memory cells, all except the first memory access are extremely fast. Most overhead that comes along with the first access need not be repeated for the rest of the cache line. This performant technique of memory access is called *cache bursting*. In order to take advantage of burst accesses in WCET computations, we propose the following technique. First consider an example in form of a control flow graph. At program point 2 we know that register $\mathbf{x}_1$ is 2^5-aligned. The edge between program point 5 and 6 contains a memory access of 4 bytes specified by the value of register $\mathbf{x}_3$.

Example 16. burst access



Summarising, at program point 5 we have the information:

- $(2^{27}\mathbf{x}_1 = 0) \equiv (\mathbf{x}_1 \mod 2^5 = 0)$
- $(\mathbf{x}_2 = \mathbf{x}_2 + 4) \mod 2^{32} \equiv 0$
- $(\mathbf{x}_3 = \mathbf{x}_1 + \mathbf{x}_2) \mod 2^{32} \equiv 0$

Since each memory operand which misses the cache causes a time penalty, it is necessary to localise 32−byte alignment within our analysis. From affine equality relation analysis the property $\mathbf{x}_3 = \mathbf{x}_1 + \mathbf{x}_2$ and $2^{27}\mathbf{x}_1 = 0$ can be deduced. When unrolling of loops and recursive functions is performed and the concrete values for registers are available, it is obvious which memory accesses are 2^5-aligned and thus are a cache miss. For the precise handling of burst accesses it is sufficient to perform partial unrolling of loops. In case of example 16, where we examine a 4-byte memory access within a loop, the loop has to be unrolled maximally 8 times — because for a word-aligned memory access 7 memory accesses within the loop are extremely fast, whereas the 8-th access will result in a cache miss and thus cause the loading of a new cache line.

Now, consider the following example, where the single array cells are accessed using pointer arithmetic:

Example 17. cache bursting

```
int main(){
  int *p;
  int arr1[16],arr2[16],arr3[16];
  int i, j;
  if (j > 100)
    p = arr1;
  else if (j > 0)
    p = arr2;
  else
    p = arr3;
  i=0;
  for(j=0;j<16;j++)
    i= i +*(p++);
  if (*p < 0)
    return 1;
  return 0;
}
```

```
//i= i +*(p++);
21:  lwz     r9,16(r1)
22:  lwz     r9,0(r9)
23:  lwz     r0,12(r1)
24:  add     r0,r0,r9
25:  stw     r0,12(r1)
26:  lwz     r9,16(r1)
27:  addi    r0,r9,4
28:  stw     r0,16(r1)
29:  lwz     r9,8(r1)
30:  addi    r0,r9,1
31:  stw     r0,8(r1)
32:  lwz     r0,8(r1)
33:  cmpwi   cr7,r0,15
34:  ble     cr7,0x21
```

Here, we are interested in alignment properties of the single memory accesses for C-construct **(p++)* within the *for*-loop, i.e. instruction 22 in assembly code. Since we examine a 4-byte memory access, this loop has to be unrolled maximally 8 times, when checking for 2^5 aligned memory accesses. Then, we try to find alignment properties for every memory access $0(r9)$ after unrolling, as the following table for example 17 illustrates:

unrolling	**address equalities** mod 2^5	**address difference** mod 2^5
0	$2^{27}r9 = 2^{27}r1 + 5 \cdot 2^{29}$	–
1	$2^{27}r9 = 2^{27}r1 + 3 \cdot 2^{30}$	4
2	$2^{27}r9 = 2^{27}r1 + 7 \cdot 2^{29}$	8
3	$2^{27}r9 = 2^{27}r1$	12
4	$2^{27}r9 = 2^{27}r1 - 7 \cdot 2^{29}$	16
5	$2^{27}r9 = 2^{27}r1 - 3 \cdot 2^{30}$	20
6	$2^{27}r9 = 2^{27}r1 - 5 \cdot 2^{29}$	24
7	$2^{27}r9 = 2^{27}r1 - 2^{31}$	28

For this purpose, we assume the address equality in the 0th step of unrolling as *reference address equality*. Next, we compute the difference by subtracting each address equality, obtained in every step of unrolling 8 times altogether, from this reference address equality. The constant differences we obtain (third column of the table above) indicate that memory is accessed in steps of 4 bytes. A difference c means that this address with respect to the reference address is equal to c modulo 2^5. Since we evaluate the equality relations with respect to the modulus 2^5, we identify that every 8th memory access within the *for*-loop will result in a cache miss. Additionally we are able to infer the alignment factor with respect to a given cache line.

Inferring alignment properties is an important aspect in cache analysis, because this information contributes much in classifying memory accesses as cache miss or hit. Since the analyses, described in this paper, were conceived within the project *SuReal* [1], the main aim consisted of yielding quite precise information about the execution properties of an executable to improve the cache behaviour prediction [12].

6 Experimental results

The theoretical approach presented in this paper was implemented within the WCET-framework in PAG [17,12]. We conducted a test series on a $2,2$ GHz Opteron machine equipped with a physical memory of 16GB. Our analysis providing affine equality relation and interval information for processor registers and memory locations quickly terminate in the range of a few seconds. Only program edn takes much more time and memory compared to the other example programs. However, it makes extensive use of local arrays, reference parameters and pointer arithmetic. All the example programs are compiled with optimisation level zero for the PPC architecture. The following table shows some practical results of our prototypical implementation.

Program	Procs	Instr	MemAcc	Unknown	Locals	Globals	PosUnal	Time	MemUse
prime	11	250	106	4	53	–	1	5.27sec	358MB
edn	16	1153	476	78	379	5	62	2024.53sec	3276MB
standard	7	116	48	–	17	–	–	0.91sec	93 MB
switch	1	93	28	1	29	–	–	1.66sec	74MB
mod	1	33	16	2	16	–	2	0.97sec	70MB
brake	6	212	112	12	44	17	–	4.54sec	382MB
top	14	1822	827	4	17	–	–	0.96sec	0,06MB

Table 2. Some results of our test programs

Within this table we specify:

1. the number of procedures $Procs$ and instructions $Instr$
2. the number of memory access instructions $MemAcc$,
3. the number of memory accesses $Unknown$ our analysis was not able to classify,
4. the number of possible candidates for local $Locals$ and global variables $Globals$,
5. the number of memory accesses $PosUnal$ which may be unaligned,
6. the time $Time$ and memory requirements $MemUse$ of our analysis run.

Our test programs are available online http://www2.in.tum.de/~flexeder/tests.
Here, we present an extract from our test suite. The first two programs $prime$ and edn are example programs from the WCET suite within the framework PAG. The program $standard$ is generated from C code which conforms to the characteristics as specified in section 2, where only local and global variables and global arrays occur. Programs mod and $switch$ are in order to test alignment properties and finally the two programs $brake$ and top generated for our SuReal demonstrator [1].
When we consider only assembly code conforming to these characteristics (section 2) all the memory accesses to local and global variables were precisely identified, as e.g. the table entry for program $standard$ specifies. This involves that the alignment property of memory accesses can be precisely classified as *aligned* or *not aligned*.

In case of programs mod and $switch$, we also take the concept of local arrays and pointers into account. In case of mod there are two memory accesses marked as *possibly unaligned*.

```
...
int *p, i;
for (i=0; i<10; i++, p++)
     *p = i;
 ...
```

We obtain this imprecise result since there is a *store*-instruction to an unknown memory location. There, we discard all information about the local variables so that even the alignment information gets lost.

In presence of the concept of pointers and pointer arithmetic, our approach is not able to precisely handle reference parameters. There all information about the local variables gets lost. As in case of an unknown target address of a *store*-instruction and passing a pointer as parameter to a function call, we destroy all information about the local variables to provide a safe analysis.

With the techniques of introducing an artificial stack pointer register and investigating escaping stack addresses, we can verify if the assembly code to analyse conforms to the specified characteristics (refer to section 2). Given the following assembly code, our analysis issues a warning that two stack addresses may escape.

Example 18. pointer arithmetic

```
01: stwu   r1,-32(r1)
02: mflr   r0
03: stw    r0,36(r1)
04: addi   r0,r1,8
05: mr     r3,r0
06: bl     0x17
07: lwz    r0,8(r1)
08: stw    r0,12(r1)
09: addi   r0,r1,12
10: mr     r3,r0
11: lwz    r0,36(r1)
12: mtlr   r0
13: addi   r1,r1,32
14: blr
```

For the assembly code to the left the variable identification mechanism inferred local variables $\_i$ at stack address $r1 - 24$ and $\_j$ at stack address $r1 - 20$. At instruction 05 the stack address $r1 - 24$ is saved in parameter passing register $r3$ and subsequently a function is called. Furthermore, at instruction 10 the stack address $r1 - 20$ is written into the return register $r3$ and is thus marked as escaping address as well. For these two instructions, our analysis issues a warning that the address of local variable $\_i$ and $\_j$ may escape and thus the assembly code violates our requested coding conventions.

Instead of issuing a warning when the callee modifies the stack level of the caller, we include this modification into our further analysis. Then, this allows for a precise classification of local variables.

Now, if we compare our results with the *value analysis* specified by AbsInt [12], for the example programs mod and $switch$, we obtain almost the same results. However, our approach is not dependent on a precise assumption on the address of the stack pointer register. Thus, if the value of the stack pointer is not known at all, our approach is able to yield at least alignment information and is able to classify memory accesses as stack address or global variable.

Concluding, we want to remark the major limitations of the approach presented here. It is not able to precisely handle local arrays and reference parameters. Since in case of a local array, we introduce a new variable for the memory location denoting the start address of the array, we may loose precision due to referencing this array. The non-standard coding conventions for parameter passing require the effect approach to yield reasonable results at all.

7 Conclusion

In this paper we presented an interprocedural analysis of executables for safety-critical real-time applications. We restrict our analysis to assembly code which contains only local and global variables and is free of the concept of pointers and address arithmetic. This is not uncommon, since e.g.s the SCADE[10] code generation guidelines [9] for safety-critical software development for avionics prove. Our approach of analysing unoptimised assembly code, using full linear algebra [21], is able to identify candidates for local and global variables and supplies information about alignment properties. We presented an analysis for inferring all valid affine equality relations between the processor registers of the executable. These relations precisely describe address expressions. The majority of address expressions which occur in real-time applications results from accesses to local variables. To establish reliable statements about local variables, it is necessary to track stack pointer modifications. For this purpose we presented the concept of introducing an artificial stack pointer. In the case of unoptimised assembly code it is indispensable to extend a conventional interval analysis to obtain precise results. For that purpose we keep track of affine relations between processor registers and memory locations. This allows us to infer intervals for single memory locations by back propagating information from processor registers to memory locations. To achieve an improvement of the precision of our analyses we follow the reduced product approach, where the information of one analysis is used to improve the precision of the other analysis. Furthermore, in this paper we presented techniques to verify the absence of pointer arithmetic and dynamic stack allocation. For intervening the area of WCET computation on executable code, we propose an alignment analysis, because the alignment of memory accesses has a crucial impact on their performance. With this information about the contents of memory locations and processor registers and their correlation, the cache analysis of [2] can be improved by deciding whether a memory access will result in a cache miss or hit. Within the project *SuReal* [1], the assembly code analysis proposed in this paper was implemented within the framework PAG [17]. Furthermore, for our approach it is not necessary to deal with contexts and a call-string length greater than zero. Compared to the congruence analysis of AbsInt within the WCET framework which relies on Granger‘s [14] congruence domain our analysis yields almost the same precise results. Applying these techniques to real-world safety-critical real-time applications seems to provide reasonable results in reasonable runtime, as section 6 illustrates.

References

1. Sicherheitsgarantien Unter REALzeitanforderungen. `http://www.sureal-projekt.org/`.
2. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.
3. G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. Technical report, 2005.
4. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, pages 1–28, 2007.
5. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin, Madison,WI,USA, August 2007.
6. S. Basumallick and K. Nilsen. Cache issues in realtime systems. 1994.
7. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
9. B. Dion. How to meet EUROCAE ED-12B / RTCA DO-178B international software safety regulation for airborne systems while reducing development cost, 2007.
10. F.-X. Dormoy and E. Technologies. *SCADE 6 A Model Based Solution For Safety Critical Software Development*, 2008. `http://www.esterel-technologies.com/technology/WhitePapers/`.
11. C. Ferdinand. Worst case execution time prediction by static program analysis. *ipdps*, 03, 2004.
12. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 469–485. Springer-Verlag, 2001.
13. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.*, 17(2-3), 1999.
14. P. Granger. Static analysis of arithmetical congruences. volume 30, pages 165–199, 1989.
15. IBM. *PowerPC Architecture - The official manual for the PowerPC architecture. Three parts: instruction set architecture, virtual environment architecture, and operating environment architecture, IBM book number SR28-5124-00*. 1993.
16. S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis technique for RISC processors. pages 97–108, 1994.
17. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
18. GrammaTech, Inc. *CodeSurfer*. `http://www.grammatech.com/products/codesurfer/`.
19. F. Mueller. *Static cache simulation and its applications*. PhD thesis, Tallahassee, FL, USA, 1995.
20. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. *SIGPLAN Not.*, 39(1):330–341, 2004.
21. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

22. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, pages 119–132, 1999.
23. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111. ACM, 2006.
24. T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *APLAS*, pages 212–229, 2005.
25. RTCA. DO-178B software considerations in airborne systems and equipment certification, 1992.
26. J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44. ACM, 1999.
27. F. Semiconductor. MPC755- RISC Microprocessor Hardware Specification, MPC755EC, Rev.8, 2006. `http://www.freescale.com/files/32bit/doc/data_sheet/MPC755EC.pdf?fpsp=1`.
28. S. Sobek and K. Burke. Power PC Embedded Application Binary Interface (EABI): 32-bit implementation, 2004. `http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf?fsrch=1`.