Multi-architecture Value Analysis for Machine Code*

Hugues Cassé, Florian Birée, and Pascal Sainrat

surname@irit.fr
Université de Toulouse
Institut de Recherche en Informatique de Toulouse (IRIT)
118 Route de Narbonne, F–31062 TOULOUSE CEDEX 9, France

Abstract

Safety verification of critical real-time embedded systems requires Worst Case Execution Time information (WCET). Among the existing approaches to estimate the WCET, static analysis at the machine code level has proven to get safe results. A lot of different architectures are used in real-time systems but no generic solution provides the ability to perform static analysis of values handled by machine instructions. Nonetheless, results of such analyses are worth to improve the precision of other analyzes like data cache, indirect branches, etc.

This paper proposes a semantic language aimed at expressing semantics of machine instructions whatever the underlying instruction set is. This ensures abstraction and portability of the value analysis or any analysis based on the semantic expression of the instructions.

As a proof of concept, we adapted and refined an existing analysis representing values as Circular-Linear Progression (CLP), that is, as a sparse integer interval effective to model pointers. In addition, we show how our semantic instructions allow to build back conditions of loop in order to refine the CLP values and improve the precision of the analysis.

Both contributions have been implemented in our framework, OTAWA, and experimented on the Malärdalen benchmark to demonstrate the effectiveness of the approach.

1998 ACM Subject Classification F.3.2 Program analysis

Keywords and phrases machine code, static analysis, value analysis, semantics

Digital Object Identifier 10.4230/OASIcs.WCET.2013.42

1 Introduction

Safety of critical embedded real-time applications needs to be verified in order to avoid catastrophic issues. This verification concerns not only functional features but also non-functional ones like temporal properties. The Worst Case Execution Time (WCET) is an important element of time properties. Its computation by static analysis is required at the machine level to get confident results.

Some tools, like OTAWA [3], provide a generic framework to perform these kinds of analyses whatever the underlying architecture is, including programming and execution models. In this article, we show how to adapt static analysis of data flow analysis to the machine code. More precisely, we have adapted and improved the circular-linear representation of integer values of [9][8] to any machine code using an architecture-independent language while maintaining fast convergence for the fixpoint computation.

* The research leading to these results has received funding from ANR under grant ANR-12-INSE-0001.

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 42–52

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Such an analysis is utterly important as its results can be used in the implementation of a lot of other analyses concerning control or data flow. For example, it can be used to compute the targets of indirect branches (pointer call or optimized `switch` implementations using indirection tables) or to identify statically code that is dynamically dead (typically in library functions where some conditions are always false because of the call context). Data flow uses include the analysis of data caches, of array accesses (bounding of the array range in memory), of the stack size (very important in embedded applications with small memory sizes) and infeasible paths by providing information on the program conditions.

This article is divided in 5 sections. In the first one, we present the abstraction of the machine language and its application to Abstract Interpretation (AI). The second section presents the CLP representation of values and a proposed refinement based on the decoding of branch conditions. Then, Section 3 gives the results of the experimentations and we provide in Section 4 a comparison with existing value analyses. The last section concludes the paper.

2 Value Analysis at Machine Code Level

In this section, we first present the abstraction of the machine instructions and how to use it to build an AI.

2.1 Independent Machine Language

As critical embedded real-time systems are running on very different architectures, OTAWA provides an abstraction of the machine instruction semantics. This avoids (a) to tie the analysis to a particular instruction set and (b) to focus only on the more useful part of the instruction behaviour to analyze the computation of integer values and addresses.

Table 1 Semantic Instructions.

Instruction (I)	Semantics ($update_I$)
`add` d, a, b	$d \leftarrow a + b$
`sub` d, a, b	$d \leftarrow a - b$
`shl` d, a, b	$d \leftarrow a \ll b$
`shr` d, a, b	$d \leftarrow a \gg b$
`asr` d, a, b	$d \leftarrow a \gg_+ b$
`set` d, a	$d \leftarrow a$
`seti` d, i	$d \leftarrow i$
`scratch` d	$d \leftarrow \top$
`cmp` d, a, b	$d \leftarrow a \sim b$
`store` d, a, t	$M_t[a] \leftarrow d$
`load` d, a, t	$d \leftarrow M_t[a]$
`if` c, a, i	$if\ a \neq c\ then\ ic \leftarrow ic + i$
`branch` d	$pc \leftarrow d$
`cont`	stop interpretation

$d, a, b \in Registers \cup Temporaries$
$i, s \in \mathbb{Z}$
$\ll$, $\gg$: logicial shift left, right
$\gg_+$: arithmetical shift right,
$\sim$: comparison,
$c \in \{=, \neq, <, \leq, >, \geq, <_+, \leq_+, >_+, \geq_+\}$
that is, signed and unsigned comparators
pc: processor program counter,
ic: counter of semantic instrucions,
$M_s[a]$: memory cell of address a of type t
$t \in \{\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}, \mathbb{N}_8, \mathbb{N}_{16}, \mathbb{N}_{32}, \mathbb{N}_{64}\}$
where k in $\mathbb{Z}_k$, $\mathbb{N}_k$ is the number of bits
$\top$: undefined value

Unlikely to an RTL (Register Transfer Language) language, it is designed to make the static analyses easier and faster. Especially, it avoids actions internal to the microprocessor that are neither relevant, nor supported by the value analysis. This language, presented in Table 1, is composed of very simple a-la RISC instructions working either on machine

registers, or on temporaries. As some instructions may be very complex, one machine instruction may match a block of semantic instructions.

The semantic instructions set have been designed to be minimal and canonical and, therefore, to make their support in analysis easier: there is only one instruction performing a semantic function. Consequently, there are only addition, subtraction, shift and move operations. As our goal is only to support arithmetics on integer and particularly on address computation, we have included neither multiplication, nor division. These operations, often on powers of 2, are implemented more efficiently as shifts. In the same way, we have no equivalent of bit-to-bit operations like NOT, AND or OR as they do not produce precise results on interval analysis. The special instruction `scratch(r)` is needed to cope with these limitations: it means that the variable r is modified in a way that cannot be described by the semantic language. Indeed, there are always machine operations too atypical to be supported but a sound static analysis requires to mark r as modified even if the exact value cannot be expressed.

As a machine instruction is usually translated into a semantic instruction block, temporary variables are needed to pass computation intermediate results all along the block. They may take place of machine register in the semantic instructions but their life is bounded to the machine instruction block.

Another important concept to support is the conditional execution of some semantic instructions. In a machine instruction set, this is performed by using a comparison instruction followed by a branch instruction to modify the control flow accordingly. This scheme is currently supported by the semantic instructions but with some limitations.

First, there is a comparison instruction `cmp` that compares two values and stores the result in a target register (usually the status register of the underlying architecture). Then the comparison result is used by an `if` instruction. If the comparison from register a is equal to the c condition argument, the semantic instruction execution continues. Otherwise, the i following instructions are skipped and the execution continues just after them. An important outcome of such a structure is that the instruction block can exhibit several execution paths but no loop. This is an important property because the fixpoint computation induced by loops needs expensive computation time: several analyses are performed on the loop body.

`cont` is the second and last instruction handling the semantics control flow: it stops the execution of the current path. It must be noticed that a `cont` is implicitely assumed at the end of a block. The `if` and `cont` handle the control flow of the semantic instructions but there are also an instruction to represent control at the machine instruction level. The `branch` instruction informs that the machine control will change according to the address found in its arguments An important point to keep in mind is that this instruction does not modify the execution flow of the semantic instructions in the block: this one continues until the end of the block. It just denotes control flow changes in the machine instructions, i.e. modification of the PC.

Finally, `load` and `store` allow to load from, or store to, the memory. The first argument is the handled value while the second contains the address. The last one is the size of handled data in bytes.

This language makes easy and straight-forward the translation of most machine instructions but it may require more work when processing instructions as complex as multiple load-store to memory. An intermediate special computation phase is required but benefits from the instruction arguments, constant in the instruction code at the generation time. For example, a particular multiple-load instruction in memory gives the precise list of loaded registers and we can generate as many `load` semantic instructions as required. This genera-

Algorithm 1 `lmw` r_d, $k(r_a)$

$b \leftarrow [seti(t_1, k); add(t_1, t_a, t_1);$
$seti(t2, 4)]$
for $i \leftarrow d$ *to* 31 **do**
$b \leftarrow b :: [load(r_i, t_1, 4); add(t_1, t_1, t_2)]$
end for

```
seti(t1, 0)
add(t1, r1, t1)
seti(t2, 4)
load(r29, t1, 4); add(t1, t1, t2)
load(r30, t1, 4); add(t1, t1, t2)
load(r31, t1, 4); add(t1, t1, t2)
```

Listing 1 lmw r29, 0(r1).

tion is performed only once per instruction and makes the static analysis faster: semantic instruction blocks are simpler than the full translation inducing loops. This is illustrated by Algorithm 1 that shows the generation of semantics instructions list (between '[' and ']') for the PowerPC `lmw` instructions: starting from address $r_a + k$, it loads registers from r_d to r_{31}. As d, a and k are constant at the translation time, a particular instantiation, as shown at the right, just gives a sequence of loads.

2.2 AI with Semantic Instructions

The semantic instructions have been designed to promote static analyses and more particularly AI. AI [5] analyzes a program by abstracting the state S along the different execution paths. With machine code, AI is often performed on the Control Flow Graph (CFG), $G = V \times E$, where the vertices V represent Basic Blocks (BB), a block of consecutive instructions executed together, and edges, $E = V \times V$, the control flow between BB.

Algorithm 2 CFG Interpretation

$wl \leftarrow \{v_0\}$
while $wl \neq []$ **do**
$v_i, wl \leftarrow wl$
$s \leftarrow update(v_i,$
$join(\{s_j / (v_j, v_i) \in E\}))$
if $s \neq s_i$ **then**
$s_i \leftarrow s$
$wl = wl \cup \{v_j / (v_i, v_j) \in E\}$
end if
end while

Algorithm 3 $update([I_0, I_1, ..., I_{n-1}], s_0)$

$s_r \leftarrow \bot$; $wl \leftarrow \{(0, s_0)\}$
while $wl \neq []$ **do**
$(i, s), wl \leftarrow wl$
if $i \geq n$ **then**
$s_r \leftarrow s_r \cup s$
else if $I_i = [\![if(c, a, d)]\!]$ **then**
$wl \leftarrow wl \cup \{(i+1, s), (i+d, s)\}$
else
$wl \leftarrow wl \cup \{(i+1, update_I(I_i))\}$
end if
end while

Algorithm 2 is a common implementation of the AI on the CFG. $v_0 \in V$ is the entry vertex of the CFG and the $s_i \in S^\#$ are the abstractions of the real state at the different program points. $S^\#$ is often a lattice with a smallest element, $\bot$, and greatest element, $\top$. $\bot$ is the initial value of s_i except for v_0 whose initial value is $\top$, that is, the more inaccurate value taking into account any possible state before the program execution. Algorithm 2 just ensures that the computation converges to a fixpoint, that is, the maximum of all possible values. This property is ensured by the existence of the lattice and the monotonicity property of any function handling the state.

This computation requires two analysis-specific functions. $update(v, s)$ emulates the effect of a basic block v on a state s while $join(s_1, s_2, ...)$ allows to combine different states coming from different paths. The semantic instructions are only used in the $update(v, s_0)$

function. For each machine instruction, the block of semantic instructions B is interpreted according to Algorithm 3. The different execution paths are supported with a working list wl containing pairs composed of the index of the current instruction and the current state. $update_I$ implements the computation effect on a state as shown in Table 1. Different execution paths are created when a `if` instruction is found: two pairs are pushed in wl for each possibility. When all execution paths have been computed, the resulting states are joined in s_r. As one may observe from Algorithm 3, the semantic instruction analysis is simple and straight-forward and re-uses directly the operators defined by the AI.

3 Proof of Concept: CLP analysis

For experimentation purpose, the semantic language has been used to implement a Circular-Linear Progression (CLP) analysis [9]. To push our semantics model even further, an analysis of conditions has been 0developed and applied to the CLP to tighten the precision.

3.1 Circular-Linear Progression Analysis

The CLP is an abstraction of integer values represented by a tuple (l, δ, m) denoting the set $\{n \in \mathbb{Z}\ s.t.\ n = l + \delta i\ \wedge\ 0 \leq i \leq m\}$, where $l \in \mathbb{Z}(n)$, $(\delta, m) \in \mathbb{N}(n)^2$. l is the base of the set, δ the increment, and m the count of increments on l to get the last point $l +_n \delta \times m$. The addition is performed on n bits (modulo 2^n), inducing a circularity on CLP and mimics the integer behaviour on the real hardware. The abstraction of a value k is easily obtained by the singleton $(k, 0, 0)$. while the top element $\top$ (a CLP that contains all possible values) is $(l, 1, 2^n - 1)$.

Performing CLP analysis on the machine code requires to abstract the semantics instructions on the CLP domain. For sake of brievity, only the `add` is given below but details on other operations can be found in [9]:

- $\{l_1\} + \{l_2\} = (l_1 +_n l_2, 0, 0)$
- $(l_1, \delta_1, m_1) + (l_2, \delta_2, m_2) = (l_1 +_n l_2, g, m_1 \frac{\delta_1}{g} + m_2 \frac{\delta_2}{g})$ where $g = \gcd(\delta_1, \delta_2)$.

The CLP allows to define the abstract states of the machine as a map from registers R and memory addresses A to CLP values, that is, $S : (R \cup A) \to clp$. AI operators are now defined by $U_S : V \times S \to S$ (update) and $J_S : S \times S \to S$ (join). J_S is naturally derived from the join function on the CLP , applied on the values assigned to registers and addresses.

The update function, U_S, is implemented as presented in the previous section. The abstract state S is applied to each machine instruction and, therefore, to each execution path of the semantics instructions. CLP values of the registers and of the memory are read or written by getting or setting them in the machine abstract state S.

3.2 Widening Function

To converge faster to a fixpoint, a widening function, $\nabla : S \times S \to S$, is useful. The example in Listing 2, a simple loop computing the sum of the elements of an array, allows to illustrate this. Listing 3 shows the translation of machine code of the loop header into semantic instructions.

At the first iteration of the loop, $p_0 = @t$, $@t$ being the actual address of array `t` in memory. At the second iteration of the loop, $p_1 = p_0 + 4 = @t + 4$. The widening ∇ is applied to these two CLP: $(@t, 0, 0) \nabla (@t + 4, 0, 0) = (@t, 4, 2^n/4 - 1)$. The resulting CLP is sound (it contains all possible values) and fast to obtain, but not very precise: next paragraph help to fix this.

```
int t[10];
int *p,int *q;
int s = 0;
q = t;
p = q;
while(p - 10 <= q)
{
        s += *p;
        p++;
}
```

Listing 2 Example of a simple C loop.

```
; r0 = 0x7fc4 (variable q)
seti t2,0x10
add t1,r31,t2    ; t1 = 0x7fc0
load r9,t1,0x4   ; r9 is p
seti t1,-0x28
add r9,r9,t1     ; r9 <- r9 - 40
cmp r71,r9,r0
if gt,r71,0x1
cont
seti t1,0x9c
branch t1
```

Listing 3 Translated Loop header.

3.3 Condition Filtering

In the example of Listing 2, we get the state $p = (@t, 4, 2^n/4 - 1)$, and a condition equivalent to $p - 40 > q$ ($10 \times sizeof(int)$ for a 32-bit machine). The edge remaining in the loop, taken if the condition is false, should provide as input state $p = (@t, 4, 2^{n-2} - 1)$ filtered by $p - 40 \leq q$ (inversed condition), i.e. $p = (@t, 4, 10)$. This would give an accurate state for the values inside the loop if we are able to build such a filter.

Unlike a condition written in a high-level programming language, the machine code does not provide a well-identified single expression for the loop condition. Instead, we must rebuild the link between variables (either registers or values in the memory) involved in the condition. In addition, one may remark that the same variable can be stored in many places, as registers or memory locations at the same time during the execution. Therefore, building the condition only on the register used in the comparison would make our approach very ineffective because it would ignore the aliasing existing between registers and memory. This is illustrated in Listing 3: the variable p is stored at the memory address `0x7fc0`, then loaded in $r9$. $r9$ is used again to carry out the result of $p - 40$. So the condition analysis must return two filters: `@0x7fc0` $> q + 40$ and $r9 > q$.

The algorithm proposed here tries to cope with both issues: the backward traversal of the semantics execution paths allows to collect the conditions of the `if` semantic instruction and to build back the aliases existing between registers and memories. Applied to the branch instruction of the loop header, the execution paths are sorted in two categories: the branching paths ending with a `branch` instruction and the continuing paths. The execution paths are then extended with other instructions of the loop header to completely form the computed condition. Each category applies a filter to the output state s of the loop header edge it matches (taken for the branch set, untaken for continuing set). As a category may contain several paths, the filters are applied on s separately and the result is joined by J_S.

Taking a path p (either branching or not), the filters are built by recording the condition induced by the `if` instructions and by rewritting the conditions when a computation instruction is found. To represent a condition, we use a simple parenthesed tree-based language where nodes are either constants c, registers r, memory places $@a$ or binary operators $\omega(e_1, e_2)$ where $\omega \in \{add, sub, ..., eq, ne, lt, ...\}$. As shown in Algorithm 4, the rewritting is performed backward (denoted by p^{-1}) from an empty set of conditions. `set` and `load` instructions are considered as creating an alias with their destination register and the filter is duplicated to generate condition for both places. The expression $f[x/y]$ means that the register x is replaced all over the filter f by y. Notice that, if a register computation is not involved in the condition building, the replacement have no effect on the filter f.

Algorithm 4 Building symbolic expressions

$f \leftarrow \{\}$
for all $s_j \in p^{-1}$ **do**
  $f \leftarrow filter[s_j]f$
end for

With $filter$ defined by:

$$filter[\texttt{if}(r, c, \_)]f = f \ \wedge \ c(r, \overline{r})$$
$$filter[\texttt{cmp}(c, a, b)]f = f[a/r][b/\overline{r}]$$
$$filter[\texttt{set}(c, a)]f = f \ \wedge \ f[a/c]$$
$$filter[\texttt{load}(c, a, \_)]f = f \ \wedge \ f[@a/c]$$
$$filter[\omega(c, a, b)]f = f[\omega(a, b)/c] \quad (1)$$

The application of the obtained filter f to an abstract state S is quite forward on a state s. For each CLP value stored in s whose reference, register or memory, is i, each reference $i' \in f$, $i' \neq i$, is replaced by its values in s, $f[s[i']/i']$ that allows, after simplification, to get a condition of the form $\omega(i, c)$ where $\omega \in \{ne, eq, lt, le, gt, ge\}$ and $c \in$ CLP. According to the actual operator ω, a CLP c_ω is obtained and intersected with the value of i: $s[i] \cap c_\omega$. For example, in Listing 3, the continuing path gives the filter $le(sub(r_9, 0x28), r0)$. Replacing the values of r_{31} and $r0$ by the matching CLP in s and simplifying gets $le(r_9, 0x7FC4 + 0x28$ and refines the value of r_9 by $s[r_9] \cap (-2^{31}, 1, 2^{31} + 0x7FC4 + 0x28) = (0x7FC4, 4, 10)$.

The application of the CLP to our semantic instructions has shown that (1) it is feasible to support such type of static analysis, and (2) there are different ways to use the semantic instructions as in condition filtering. We can hope that the semantic instruction is a valuable abstraction of the machine code instructions. In the next section, we try to evaluate the performances of this representation.

4 Experimentation

The value analysis presented in this paper has been implemented in the OTAWA framework [3] using its own internal AI engine. The evaluation has been performed with a 3-GHz, 2GB memory Linux machine on the classic Mälardalen benchmark [1] that contains a collection of programs covering different embedded real-time domains .

4.1 Analysis Precision

This first evaluation criterium concerns the precision of the performed analysis. We are not able to estimate the actual precision of the obtained measurements in terms of difference between the real values and the analyzed ones: (a) we have no other analysis that can be taken as a reference and (b) it is impossible to have precise values as soon as a program is using external inputs. The more visible trace of imprecision is the apparition of $\top$ values in the computation. Yet, it is hard to qualify the actual source of this imprecision as being naturally produced by the AI or an outcome of the intrinsic inefficiency of our analysis. In turn, the last issue may be decomposed in two causes: the lack of expressivity of semantics instructions or the limits of the abstraction of the CLP analysis.

Whatever, the last three columns of Figure 2 show the number of non-$\top$ values obtained for different items: each column represents the ratio of non-$\top$ values on the total of values of the measured items, the bigger is the better. The *values* column displays the number of set values generated by `set` and `store` semantic instructions with an average of 42.30% of non-$\top$ values. In the absolute, the result is a bit disappointing but (1) to our knowledge, no such statistics have been published to compare with and (2) the large variation between benches (from 2.47% to 99.98%) shows that the effectiveness has a big dependency on the type of

Table 2 Analysis Execution Time.

Program	Time (μs)	Dynamic mach (i/s)	Dynamic sem (i/s)	Static mach (i/s)	Static sem (i/s)	Value Precision values (%)	addrs (%)	filters (%)
adpcm	31530	219600	495908	39861	204598	2.47	15.39	72.34
bs	1190	103361	228571	97560	141176	53.85	89.09	33.33
bsort100	2680	117537	269029	63492	110074	20.75	17.31	33.33
cnt	2590	214285	462548	50450	188030	36.17	46.51	100.00
compress	25610	85591	196134	48813	88481	8.68	27.42	49.30
cover	50040	59492	159852	156533	81434	8.66	52.94	100.00
crc	5160	230232	446511	47979	207751	10.04	79.25	86.52
duff	1100	97272	233636	271028	441818	38.46	58.70	60.87
expint	3290	115805	272644	70866	132218	47.37	71.43	80.00
fac	30	1400000	3233333	1000000	4533333	77.78	95.00	100.00
fdct	1330	958646	2051127	8627	1057142	20.98	99.42	100.00
fft1	42220	247418	485243	17231	87162	24.77	61.90	16.03
fibcall	910	98901	252747	100000	142857	74.19	100.00	100.00
fir	3330	104504	268168	45977	97897	34.48	90.50	16.00
insertsort	1070	241121	481308	34883	202803	47.37	72.22	33.33
janne_c	2360	72457	188559	93567	74576	59.52	87.21	77.78
jfdctint	1940	718556	1690721	10043	598453	6.72	91.57	100.00
lcdnum	4040	73267	177970	162162	92574	30.99	50.45	100.00
lms	14590	202604	385263	33491	125222	49.47	91.25	81.94
ludcmp	12660	138151	280489	29731	77725	33.16	82.85	29.65
matmult	4140	238647	503864	38461	145893	23.33	29.44	100.00
minver	16320	145220	306250	32911	87438	13.49	67.93	29.00
ndes	27550	167622	369546	33347	111724	52.28	74.46	74.39
nsichneu	205180	98450	209182	37425	104664	11.98	89.68	29.80
ns	8060	99503	206203	24937	29528	83.58	89.40	61.29
prime	2500	162800	362000	100737	237200	70.75	100.00	50.00
qsort-exam	5860	139761	289419	41514	150170	13.46	63.50	16.85
qurt	6320	154113	305537	61601	233386	44.81	100.00	100.00
recursion	30	800000	2033333	1714285	4066666	69.23	100.00	100.00
select	5150	136893	283300	51063	149708	19.75	66.42	36.17
sqrt	1610	91304	182608	74829	88198	50.00	100.00	100.00
statemate	48770	157227	307709	40688	109883	3.81	90.85	89.02
st	7680	213411	430989	58572	245833	10.12	19.33	100.00
ud	7630	144429	300262	37205	114285	33.59	81.25	26.72
average		242593	539705	139114	428232	34.88	72.14	67.17

the program. Some benchmarks like *crc* or *adpcm* give particularly bad results because they are performing a lot of operations unsupported by our semantics instructions, respectively, bit-to-bit and floating-point operations.

However, our analysis works better with address computation (74% on a mean, column *addrs*) and condition filtering (68% on a mean, column *filters*). The *Addresses* column evaluates the non-$\top$ addresses used in `load` and `store` instructions while the *filters* column

evaluates the number of non-$\top$ filtered conditions. The obtained results are not perfect but they meet the needs of accuracy required by subsequent analyses like data cache, control flow analysis, infeasible path, etc. As for *set values*, the worst results are obtained on benchmarks using unsupported operations of the semantic language.

4.2 Computation Time

The six first columns of Table 2 show runtime performances of the analysis. We call *static* estimation the performance evaluation based on the count of instructions found as-is in the program once loaded in memory. On the opposite, *dynamic* instructions are counted during the AI. Both estimations are different because a single instruction may be interpreted several times before reaching a fixpoint: this is particularly the case of instructions contained in loops. The left column displays the experimented program, the next one the analysis time in μs (user time mean after 1000 iterations). The following two columns show the rate in machine and in semantic instructions, for dynamic estimation and the last two columns are the same for static.

The static estimation gives an insight of the capacity of the analysis to face to a raw program. For example, the average of about 139000 instructions / s states that we can process quickly (in less than 1 second) a program of nearly 512KB of code for a PowerPC architecture or any pure 32-bit RISC architecture.

On the other hand, the dynamic performances give a more straight-forward estimation of the real rate of the analysis, 242,594 machine instructions/s and 539,705 semantic instructions/s, taking into account the structure of the program. Hopefully, both measures are quite good although there is a lot of variability according to the benchmarks. Such performances let place for improving analysis without being blocked by intractable computation time.

5 Comparison with Existing Work

A lot of work on value analysis has been done for different purposes. The foundation of abstract AI was motivated by the interval analysis $[l, u]$ of each variable on Pascal-like languages [5]. In addition to the domain, the performed analysis is also different because of the widening operation. To speed up the convergence to the fixpoint, widening and narrowing operators are applied, that requires several passes of analysis and increases the computation time. To our knowledge, although there is very few documentation on this [4], the tool ait [2] is also using such an analysis to exhibit register values at program points but is applied to machine language.

Yet, the interval of values $[l, u]$ does not fit well with address representation. They include too many false values while the addresses are usually aligned to a multiple of data sizes and cause the union of too many false data and reduce the precision. Ermedhal et al in [6] improve this using a modulo analysis to identify more precisely values inside an interval. Yet, their domain is more complex than ours, $([l, u], (i, c))$ that identifies the set of values matching $[l, u] \cap \{i + n \times c, n \in \mathbb{Z}\}$. Moreover, this domain makes the analysis more costly in computation time.

S. Rathijit [8][9] simplifies a lot this representation using only three integers, (l, u, δ). i, the argument of Ermedhal's representation is easily replaced by finding the lowest value of the set to get a more precise l. Then, we have improved Rathijit's work to have canonical representation of these values. Indeed several representations for a same set exist because $u - l$ is not required to be a multiple of δ. Our triplet (a, δ, n) ensures the uniqueness of the representation as it forces $u = a + n \times \delta$. This property makes easier some operations of

the value arithmetics like equality tests. In addition, unlike Rathijit that seems to support only ARM, our machine code abstraction, avoids to tie our analysis to a specific machine instruction set and we are using the conditions to accelerate convergence of fixpoint of the analysis and improve the precision in presence of selections.

In the domain of semantics languages, ALF [7] is supported by several compilation and WCET tools. It allows to represent completely a program while our language is just an overlay on the CFG. More comprehensive and more expressive than our semantics instructions, it seems to be more memory- and time-consuming for analyses. Architecture Description Languages like SimNML, Lisa, etc are also good candidate to express semantics of machine code. But we think that their granularity would make the analyses too costly and that some concepts are too hard to extract: for example, in a microprocessor, the comparison result is represented as a set of bits obtained by a combination of operand bits whose usage in analysis is not straight-forward.

6 Conclusion

The contribution of our paper is twofold. First, we propose a language to abstract usual machine code semantic and show how it may be involved in abstract AI. The overall outcome is that the analyses based on this language becomes portable on any microprocessor supported by our semantic language. For example, in the OTAWA framework, it has been already successfully used to describe several RISC instruction sets like PowerPC. We plan to quickly apply the semantics instructions to Sparc, TriCore and possibly to the x86 instruction set.

Second, we have improved existing value analysis, based on CLP domain, for speed of computation. The changes include a reformulation of the domain leading to canonical values and therefore simplification of operation abstraction. The second improvement concerns the use of conditions to speed up the convergence of fixpoint computation.

The resulting analysis rate is quite fast, which will be valuable in the future in order to fix medium precision results. We think that the precision problem is not inherent to the value analysis algorithm but, instead, to the limitation of the semantic language. So, we plan to extend it with bit-to-bit operations, integer multiplication and division as well as floating-point operations. The latter extension cannot be done in the frame of CLP because the notion of modulo does not fit well with float value: we have to fall back to usual interval analysis. This means a state abstraction with heterogenous content.

Finally, we would like to exploit the results of this value analysis to extend the OTAWA framework. The semantic language has already been used in OTAWA to implement data cache analysis and stack analysis but we can exploit the value analysis results to improve control flow analyses like detection of infeasible paths, dynamically dead code, switch decoding, indirect pointer call analyzes. As the latter analysis does not fit well with CLP, we have also to introduce set of addresses in our domain and it remains to identify which values should be CLP and which ones should be sets. In a more generic way, we have to replace our naive implementation of state to improve our value representation to support heterogeneous data, to maintain fast computation and to waste as less memory as possible.

References

1 Mälardalen benchmarks. http://www.mrtc.mdh.se/~projects/~wcet/~benchmarks.html.

2 ait tool, 2005. http://www.absint.com/ait/.

3 C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

4 C. Ferdinand, R. Heckmann, and D. Kästner. Static Memory and Timing Analysis of Embedded Systems Code. In *Proceedings of the IET Conference on Embedded Systems at Embedded Systems Show*, 2006.

5 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*, 1977.

6 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation and Invariant Analysis. In *7th International Workshop on Worst-Case Execution Time Analysis*, 2007.

7 J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF – A Language for WCET Flow Analysis. *WCET'09*, 30 June 2009.

8 S. Rathijit and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. 2007.

9 S. Rathijit and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, 2007.