Validation of Assembler Programs for DSPs: A Static Analyzer

Matthieu Martel
Commissariat à l'Energie Atomique
LIST-DTSI-SOL
CEA F91191 Gif-Sur-Yvette Cedex, France
matthieu.martel@cea.fr

ABSTRACT

Digital Signal Processors are widely used in critical embedded systems to pilot low-level, often critical functionalities. We describe a static analyzer based on abstract interpretation and designed to validate industrial assembler programs for a DSP. The validation consists of guaranteeing the absence of runtime errors such as incorrect memory accesses and of tracking the sources of inaccuracies introduced by floating-point computations. Our first contribution is a new static analysis for relocatable assembler programs able to cope with dynamically computed branching addresses. Our second contribution is the analyzer itself and its graphical interface which helps the user to understand the numerical inaccuracies.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*Formal methods, Validation*; F.3.1 [**Logics and meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants, Mechanical verification*; G.1.0 [**Numerical Analysis**]: General—*Computer Arithmetic, Error Analysis, Stability and Instability*

General Terms

Algorithms, Design, Experimentation, Reliability, Verification

Keywords

Abstract Interpretation, Floating-Point Numbers, Numerical Accuracy

1. INTRODUCTION

Digital Signal Processors (DSPs) are widely used in critical embedded systems like airplanes or automobiles. They pilot low-level, often critical functionalities such as driving a device or acquiring and processing numerical data. These DSPs are programmed either in C or assembler to have a direct access to the interfaces, buses, etc. of the systems. An important characteristic of DSPs is that they are intended to perform numerical computations for signal processing such as digital filters or Fourier's transforms. When these computations are carried out with floating-point numbers, the roundoff errors may introduce large inaccuracies, making the resulting values irrelevant [6]. In this article we describe a static analyzer based on abstract interpretation [3, 4] and designed to validate assembler programs for the TMS320 C3X processor [14], a widely used DSP. This analyzer is intended to cope with industrial-size codes, and is currently being tested by Airbus. The validation consists of guaranteeing the absence of runtime errors such as incorrect memory accesses or type errors and of tracking the sources of inaccuracies introduced by floating-point computations using recent semantics [7, 9]. Recently, abstract interpreters have shown their ability to validate industrial C programs, for runtime error detection [2] as well as for numerical accuracy [12]. In this work, our choice of assembler stems from the fact that many DSP codes are directly written in this language and that, otherwise, the numerical computations are very sensitive to the compiler and processor. For instance, C compilers often do not respect the evaluation order of arithmetic expressions and floating-point value are stored with more digits in registers than in memory. The validation of low-level codes cope with such details.

Our first contribution is the abstract semantics of assembler codes formally defined in Section 2. We analyze relocatable programs in which the addresses are defined by labels. Because relocatable programs still need to be loaded at some absolute memory address, they contain useful information on data structures as well as on the data and the text (code) segments. Next, our analysis does not rely on a prebuilt control flow graph. Building a control flow graph for assembler programs is difficult [15] and, instead, we evaluate the branching addresses at analyze time, which yields more precise results and enables us to cope with dynamically calculated branching addresses. As a consequence, our analysis is also accurate for recursive programs. A drawback of our approach is that assemblers may differ from one processor to another, making the analysis too specific. We believe that our language is representative of many other assemblers.In Section 5, we discuss how some aspects of our work should be modified for other processors. Our second contribution is the analyzer itself, which is described in Section 4. Mostly,

PASTE'04, June 7-8, 2004, Washington, DC, USA.

we show how numerical errors can be tracked by means of the graphical interface. Because the tool uses the semantics of error series [7, 9] to track the propagation of errors in floating-point computations, we remind the reader of this semantics in Section 3.

We end this section with some bibliographic notes. Recently, abstract interpretation has been successfully applied to the calculation of Worst Case Execution Times (WCET) of binary codes [5]. These analyses differ significantly from ours in the nature of the results and also technically, since they rely on an abstract simulation of the micro-instructions of the processors and they use a pre-built control-flow graph [15]. Another interesting approach, proposed by X. Rival, concerns the validation of compiled programs. Invariants are computed in the source code and then translated to be checked on the binary code [13]. This approach (which requires the existence of a C source program) benefits from the techniques developed to analyze high level languages but complicated invariants, like these used in numerical precision [7, 9], may be difficult to translate. Finally, other work has focused on the static analysis of binary codes, mainly to ensure that mnemonics are well-typed [16, 17]. Here, the lack of relocation information makes it necessary to annotate the programs before analysis, to specify the data types.

2. THE ANALYSIS

This section introduces the abstract semantics used by our analyzer on a simplified assembler language. As shown in Figure 1, we use integer and floating-point arithmetic operations (`ADDI`, `ADDF`, etc.), the `MV` mnemonic for load and store operations, `PUSH` and `POP` mnemonics to manage the stack, a procedure call mnemonic (`CALL`) and unconditional and conditional branch mnemonics (`BR`, `BLT` and `BEQ`). `CMPI` and `CMPF` compare the operands src_1 and src_2 and set the flags `LT` and `EQ` to true or false, depending on whether $src_1 < src_2$ and $src_1 = src2$, respectively. `BLT` (resp. `BEQ`) executes the branchings if the `LT` flag (resp. the `EQ` flag) is true. `CALL` pushes the program counter register (pc) onto the stack and executes the branchings. The source and destination arguments of the mnemonics are defined by four addressing modes. For the sake of simplicity, we allow arbitrary addressing modes for all instructions, while in practice the TMS320 has some restrictions. In immediate addressing, v denotes a value, i.e. an integer, a floating-point number or a relocatable address defined by a label. The register mode enables to read or write the values of the registers. The simplified register set used in this article only contains general registers $R_0, \ldots, R_n$. In the direct mode, $*v$ denotes the value stored at the memory location v. Finally, in the indirect mode, $R_i(R_j)$ denotes the memory location $a + b$, where a and b respectively are the values of R_i and R_j. A program is made up of two sections, for data and text. The data section is made up of declarations of initialized memory locations and, in the text section, we assume that each mnemonic is encoded by exactly one memory word.

For the abstract domains, since the assembler values contain pointers, the representation of global memory states and the domains of elementary values are closely related. Firstly, we introduce our abstraction of the global memory states, assuming that the elementary values belong to an abstract domain $\mathcal{D}^\sharp$. An abstract memory state is represented by a quintuplet $\mathcal{M} = (\rho, \mu, \sigma, \varphi, pc)$ where $\rho : [0, n] \to \mathcal{D}^\sharp$ maps any register R_i, $1 \leq i \leq n$, to an abstract value, $\mu : Loc \to \mathcal{D}^\sharp$ maps any memory location to an abstract value, σ is the abstract stack defined later, $\varphi : flags \to \mathcal{B}^\sharp$ assigns to each flag an abstract boolean value $b \in \mathcal{B}^\sharp = \{\texttt{true}, \texttt{false}, \top, \bot\}$ and $pc \in \mathcal{D}^\sharp$ is the program counter register. μ records the values stored in the data segment. A location $\ell \in Loc$ is a pair (lab, i) where lab is a data label and i is an integer. (lab, i) denotes the location of the i^{th} value defined in the declaration of lab. For example, the declaration `d: .INT 1,2` makes $\mu(\texttt{d}, 0) = 1$ and $\mu(\texttt{d}, 1) = 2$. In our analyzer, if n values are allocated at label lab then attempting the access $\mu(lab, m)$, for $m \geq n$, raises an error comparable to an out of bound access to an array. The function φ maps the flags of the processor to abstract boolean values. In our simplified language, we consider two flags, `EQ` and `LT`, which are modified by `CMPI` and `CMPF`.

The domain $\mathcal{D}^\sharp$ of values is composed of integers, floating-point numbers and data and text pointers belonging to $\mathcal{I}$, $\mathcal{F}$, $\mathcal{D}_*$ and $\mathcal{T}_*$, respectively. The abstract integers belong to the domain $\mathcal{I}$ of intervals with integer bounds. The floats are abstracted by error series [7, 9] belonging to $\mathcal{F}$, (see Section 3). Let $\mathcal{L}_D$ denote the set of labels defined in the data section and let $\wp(X)$ denote the powerset of X. The abstract domain of pointers to the data section is

$$\mathcal{D}_* = \wp(\mathcal{L}_D) \times \mathcal{I} \tag{1}$$

The concretization of an abstract value $(L, I) \in \mathcal{D}_*$ is:

$$\gamma_{\mathcal{D}_*}(L, I) = \big\{(lab, i) \ : \ lab \in L, \ i \in I\big\} \tag{2}$$

The join and comparison operators over $\mathcal{D}_*$ are defined componentwise. The text pointers are defined similarly to the data pointers, using the set $\mathcal{L}_T$ of text labels instead of $\mathcal{L}_D$. Finally, the domain $\mathcal{D}^\sharp$ of abstract values is defined by: $\mathcal{D}^\sharp = \{\bot_{\mathcal{D}^\sharp}, \top_{\mathcal{D}^\sharp}\} \cup \mathcal{I} \cup \mathcal{F} \cup \mathcal{D}_* \cup \mathcal{T}_*$.

Our choice of domains, mainly to model pointers, enables the analysis to detect possible runtime errors, some of which are specific to assembler programs. The analyzer detects the read and write operations of data into the text segment, possible runtime errors comparable to the out of bound accesses to arrays in high-level languages, branchings out of the text segment, absolute references to the memory (in relocatable programs), type mismatch errors and the use of non-initialized registers. Even if some programmers may occasionallly use the preceding dangerous statements, they are error-prone in general and our validation tool reports them. Prohibited branchings even make the analyzer stop since it does not know the next control point.

The abstract semantics is given in Figure 2. $\mathcal{M}\langle pc \to mnem\rangle \mapsto \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ means that the current memory state $\mathcal{M}$ in which the program counter register pc points to the mnemonic $mnem$ leads to one new memory state $\mathcal{M}_i$ for some $1 \leq i \leq n$. A memory state $\mathcal{M}$ is a quintuplet $(\rho, \mu, \sigma, \varphi, pc)$ as defined previously. $\mathcal{M}\langle loc\texttt{:=}v\rangle$ denotes the memory state $\mathcal{M}$ in which the location loc is set to v.

The operands of the mnemonics are addressing modes whose abstract semantics must be defined. We treat source and destination operands differently. The semantics $[\![a]\!]_s$ of a source operand is:

$$[\![v]\!]_s = v \qquad [\![R_i]\!]_s = \rho(R_i) \tag{3}$$

$$[\![*v]\!]_s = \bigcup_{\ell \in \gamma_{\mathcal{D}_*}(v)} \mu(\ell) \qquad [\![R_i(R_j)]\!]_s = \bigcup_{\ell \in \gamma_{\mathcal{D}_*}\left(\rho(R_i) +_{\text{int}} \rho(R_j)\right)} \mu(\ell) \tag{4}$$

For the direct and indirect modes, the argument is evaluated yielding an abstract label $\ell^\sharp \in \mathcal{D}_*$ and the final result joins

$$
\begin{array}{rcll}
mnem & ::= & \texttt{ADDI}\ src_1, src_2, dst \mid \texttt{SUBI}\ src_1, src_2, dst \mid \texttt{MULI}\ src_1, src_2, dst & \\
& \mid & \texttt{ADDF}\ src_1, src_2, dst \mid \texttt{SUBF}\ src_1, src_2, dst \mid \texttt{MULF}\ src_1, src_2, dst \mid \texttt{MV}\ src, dst & \\
& \mid & \texttt{BR}\ src \mid \texttt{BLT}\ src \mid \texttt{BEQ}\ src \mid \texttt{CMPI}\ src_1, src_2 \mid \texttt{CMPF}\ src_1, src_2 \mid \texttt{PUSH}\ src \mid \texttt{POP}\ dst \mid \texttt{CALL}\ src & \\
\\
src, dst & ::= & val & (Immediate) \\
& \mid & R_0 \mid R_1 \mid \ldots \mid R_n & (Register) \\
& \mid & *val & (Direct) \\
& \mid & R_i(R_j) & (Indirect) \\
\\
val & ::= & int \mid float \mid label & \\
\\
decl & ::= & \texttt{label :}\ \texttt{.INT}\ (\texttt{int}\ list) \mid \texttt{label :}\ \texttt{.FLOAT}\ (\texttt{float}\ list) & \\
\\
prog & ::= & (decl\ list)((\texttt{label})?\ mnem)\ list &
\end{array}
$$

Figure 1: Syntax of the language.

the values stored at concrete locations ℓ, for all $\ell \in \gamma_{\mathcal{D}_*}(\ell^\sharp)$. $+_{\texttt{int}}$ is defined in equations (8) and (9). For destination operands we have:

$$[\![v]\!]_d = \text{error} \qquad [\![R_i]\!]_d = \{R_i\} \tag{5}$$

$$[\![*v]\!]_d = \begin{cases} \{\mu(x) \ : \ x \in \gamma_{\mathcal{D}_*}(v)\} \text{ if } v \in \mathcal{D}_* \\ \text{error otherwise} \end{cases} \tag{6}$$

$$[\![R_i(R_j)]\!]_d = \{\mu(x) \ : \ x \in \gamma_{\mathcal{D}_*}([\![R_i]\!]_s +_{\texttt{int}} [\![R_j]\!]_s)\} \tag{7}$$

The evaluation of destination operands yields a set of location or registers. $\mathcal{M}\langle [\![m]\!]_d \texttt{:=} v\rangle$ abbreviates $\mathcal{M}\langle d_1 \texttt{:=} v, \ldots, d_n \texttt{:=} v\rangle$, for $d_1, \ldots, d_n \in [\![m]\!]_d$. In the direct and indirect modes, the number of labels of a program being finite, $\gamma_{\mathcal{D}_*}(m)$ always denotes a finite number of locations.

In Figure 2, the first rule holds for the integer operations `ADDI`, `SUBI` and `MULI`. In order to handle arithmetic operations on pointers, we allow some coercions between values of the domains $\mathcal{I}$, $\mathcal{D}_*$ and $\mathcal{T}_*$.

$$i_1 \pm_{\texttt{int}} i_2 = i_1 \pm i_2 \quad \text{if } i_1 \in \mathcal{I} \text{ and } i_2 \in \mathcal{I} \tag{8}$$

$$d \pm_{\texttt{int}} i = (L, I) \in \mathcal{D}_* \quad \text{if } d = (L, i') \in \mathcal{D}_*,\ i \in \mathcal{I} \text{ and } I = i \pm i' \tag{9}$$

A coercion rule similar to (9) holds for text pointers. The abstract interpretation of the unconditional branch `BR` yields a set $S = \{\mathcal{M}\langle pc\texttt{:=}pc'\rangle \ : \ pc' \in \gamma_{\mathcal{T}_*}([\![src]\!]_s)\}$ of possible configurations. `BLT` and `BEQ` behave like `BR` whenever the related flag is set to `true`. The next rule of Figure 2 concerns the `CMPI` mnemonic. For `CMPI`, when src_1 is compared to src_2, four new memory states combining the possible values of the flags are generated. $\mathcal{M}_{|src_1 \leq src_2}$ denotes the memory state $\mathcal{M}$ in which the values of the operands have been narrowed to the only cases making the condition true.

Our abstract stack domain is designed to behave well for loops in which the same sequence of `PUSH` and `POP` occurs at each iteration and a `PUSH` argument always has the same type. The abstract stack is described by a directed valued graph whose nodes correspond to the `PUSH` and `CALL` operations of the analyzed program. Intuitively, an edge (n_1, n_2) indicates that the `PUSH` or `CALL` related to the node n_2 may precede the one related to n_2. The abstract value attached to a node n over-approximates the abstract values pushed by the corresponding statement. Finally, our abstract stack also has a set of possible top elements.

Formally, an abstract stack σ is a pair (G, τ) where $G = (V, E, \omega)$ is a valued graph and τ a set of nodes. V has one node per `PUSH` and `CALL` statement of the analyzed program, $E \subseteq V \times V$ is the set of edges, $\omega \ : \ V \to \mathcal{D}^\sharp$ maps any node to an abstract value and and $\tau \subseteq V$ is a set of possible tops of the stack. Initially, $\omega(v) = (\bot_{\mathcal{D}^\sharp}, \emptyset)$ for any $v \in V$ and $\tau = \emptyset$. The semantics of `PUSH`, `POP` and `CALL` is given in Figure 2. In the semantics of `CALL`, $T(pc+1)$ denotes the $\mathcal{T}_*$ value pointing to the line $pc+1$ in the text section. When `PUSH` src occurs at the i^{th} line of code of the analyzed program, the argument is evaluated, yielding $v = [\![src]\!]_s$, and v is joined to the values already recorded in $\omega(i)$. Next, the current top elements of τ are added to the successors of i and τ is set to $\{i\}$. For `POP` dst, the argument is evaluated by $[\![dst]\!]_d$ and the resulting locations are set to the union of the abstract values occurring in the possible top positions of the stack. The new top of the stack is set to the union of the successors of the possible tops. Finally, the semantics of `CALL` pushes the return address as values and executes the branching. The union of two abstract stacks $\sigma_1 = (G_1, \tau_1)$ and $\sigma_2 = (G_2, \tau_2)$ is defined by: $(G_1, \tau_1) \cup (G_2, \tau_2) = (G, \tau_1 \cup \tau_2)$ where, in G, $\forall v \in V,\ \omega(v) = \omega_1(v) \cup \omega_2(v)$ and $E = E_1 \cup E_2$. The new top is the union of the tops of σ_1 and σ_2 and the levels of the stacks are joined element by element. Our model of the stack is accurate enough to analyze recursive procedures.

We now describe how widenings are performed and how the abstract environments are stored in the analyzer. For each line of code i, we assume that $\mathcal{E}(i)$ records the environment for which the mnemonic has been analyzed. The analysis then consists of analyzing states and updating $\mathcal{E}$ and a work list containing the configurations not yet examined, until the work list becomes empty. Obviously, a configuration is not analyzed again if it has already been analyzed with a greater environment. To decide when widenings need to be performed, a counter is assigned to each branching. $B(i)$ denotes the counter related to the branching of line i and it is incremented every time the mnemonic of line i is evaluated. Then a widening is performed when $B(i)$ becomes greater than a user defined constant, and $B(i)$ is reset to zero.

A widening $\mathcal{M}_1 \nabla \mathcal{M}_2$ consists of separately widening each register, memory location, and flag. The widening of stacks is described later on. Concerning the elementary values, the intervals of integers are widened in a usual way and, for pointers, no widening is performed: let $d_1, d_2 \in \mathcal{D}_*$ and $t_1, t_2 \in \mathcal{T}_*$. $d_1 \nabla d_2 = d_1 \cup d_2$ and $t_1 \nabla t_2 = t_1 \cup t_2$. Note that, since the set of data and text locations is finite, the analysis always terminates even if the abstract pointers are not widened. Let $\sigma = (G, \tau)$, $\sigma_1 = (G_1, \tau_1)$ and $\sigma_2 = (G_2, \tau_2)$ be stacks such that $\sigma = \sigma_1 \nabla \sigma_2$, $G_1 = (V_1, E_1, \omega_1)$ and $G_2 = (V_2, E_2, \omega_2)$. Then σ is defined by $\tau = \tau_1 \cup \tau_2$, $E = E_1 \cup E_2$ and $\forall v \in V,\ \omega(v) = \omega_1(v) \nabla \omega_2(v)$. The stack is widened level by level and the sets of possible top elements are joined.

Our last point about the analyzer concerns the management of an environment $\mathcal{E}(i)$ containing the abstract mem-

$$\mathcal{M}\langle pc \to \texttt{ADDI}\ src_1, src_2, dst\rangle \mapsto \{\mathcal{M}\langle [\![dst]\!]_d := [\![src_1]\!]_s +_{\texttt{int}} [\![src_2]\!]_s,\ pc := pc+1\rangle\}$$

$$\mathcal{M}\langle pc \to \texttt{ADDF}\ src_1, src_2, dst\rangle \mapsto \{\mathcal{M}\langle [\![dst]\!]_d := [\![src_1]\!]_s +_{\texttt{float}} [\![src_2]\!]_s,\ pc := pc+1\rangle\}$$

$$\mathcal{M}\langle pc \to \texttt{MV}\ src, dst\rangle \mapsto \{\mathcal{M}\langle [\![dst]\!]_d := [\![src]\!]_s,\ pc := pc+1\rangle\}$$

$$\mathcal{M}\langle pc \to \texttt{BR}\ src\rangle \mapsto \{\mathcal{M}\langle pc := pc'\rangle\ :\ pc' \in \gamma_{\mathcal{T}_*}([\![src]\!]_s)\}$$

$$\mathcal{M}\langle pc \to \texttt{BLT}\ src\rangle \mapsto \{\mathcal{M}\langle pc := pc'\rangle\ :\ pc' \in \gamma_{\mathcal{T}_*}([\![src]\!]_s)\} \text{ if } \varphi(\texttt{LT}) \sqsupseteq \texttt{true}$$

$$\mathcal{M}\langle pc \to \texttt{BLT}\ src\rangle \mapsto \{\mathcal{M}\langle pc := pc+1\rangle\} \text{ if } \varphi(\texttt{LT}) \sqsupseteq \texttt{false}$$

$$\mathcal{M}\langle pc \to \texttt{CMPI}\ src_1, src_2\rangle \mapsto$$
$$\{\mathcal{M}_{|src_1 \le src_2}\langle \varphi(\texttt{LT}) := \texttt{true}, \varphi(\texttt{EQ}) := \texttt{true}, pc := pc+1\rangle, \mathcal{M}_{|src_1 < src_2}\langle \varphi(\texttt{LT}) := \texttt{true}, \varphi(\texttt{EQ}) := \texttt{false}, pc := pc+1\rangle,$$
$$\mathcal{M}_{|src_1 = src_2}\langle \varphi(\texttt{LT}) := \texttt{false}, \varphi(\texttt{EQ}) := \texttt{true}, pc := pc+1\rangle, \mathcal{M}_{|src_1 > src_2}\langle \varphi(\texttt{LT}) := \texttt{false}, \varphi(\texttt{EQ}) := \texttt{false}, pc := pc+1\rangle\}$$

$$\mathcal{M}\langle pc \to \texttt{PUSH}\ src\rangle \mapsto \mathcal{M}\langle \omega(pc) := \omega(pc) \cup [\![src]\!]_s,\ \forall t \in \tau\ :\ E := E \cup (pc, t), \tau := \{pc\},\ pc := pc+1\rangle$$

$$\mathcal{M}\langle pc \to \texttt{POP}\ dst\rangle \mapsto \mathcal{M}\langle [\![dst]\!]_d := \bigcup_{t\in\tau} \omega(t),\ \tau := \{t'\ :\ (t,t') \in V \wedge t \in \tau\},\ pc := pc+1\rangle$$

$$\mathcal{M}\langle pc \to \texttt{CALL}\ src\rangle \mapsto \mathcal{M}\langle \omega(pc) := \omega(pc) \cup T(pc+1),\ \forall t \in \tau\ :\ E := E \cup (pc, t), \tau := \{pc\},\ pc := \gamma_{\mathcal{T}_*}([\![src]\!]_s)\rangle$$

Figure 2: Abstract semantics of the language.

ory state of the i^{th} line of code. Basically, it is too imprecise to record, for each line of code i, a single memory state $\mathcal{M}$. This stems from the fact that the abstract semantics of the comparison operators CMPI and CMPF yields many environments with different values of the flags and differently narrowed values of the arguments. For example, let us assume that a comparison occurs at the i^{th} line of code of a program. If $\mathcal{E}(i+1)$ only recorded one memory state then all the memory states generated by the comparison of line i would be merged at line $i+1$, leading to a very imprecise analysis. So the analyzer separately records the memory states for which a mnemonic has been analyzed, depending on the values of the flags. In other words, $\mathcal{E}(i, b_{\texttt{LT}}, b_{\texttt{EQ}})$ records the memory states $\mathcal{M}$ for which a program has been analyzed and such that, in $\mathcal{M}$, $\varphi(\texttt{LT}) = b_{\texttt{LT}}$ and $\varphi(\texttt{EQ}) = b_{\texttt{EQ}}$. So, four abstract environments are stored per line of code, our language using two flags. This is memory-consuming but it is necessary since, otherwise, the information inferred from comparisons is lost.

3. NUMERICAL ACCURACY

This section briefly introduces the concrete domain of error series [7, 9]. Our analyzer uses an abstraction of this domain which can be straightforwardly defined. Let $\mathbb{F}$ be the set of floating-point numbers and let $\uparrow_\circ\colon \mathbb{R} \to \mathbb{F}$ be the function which returns the roundoff to the nearest of a real number $r \in \mathbb{R}$. The TMS320 specification states that any datum d is rounded to $\uparrow_\circ (d)$. In addition, the TMS320 specification states that, like in the IEEE 754 Standard [1, 6], the elementary operations are correctly rounded, i.e.$\diamond \in \{+, -, \times\}$,

$$\forall f_1, f_2 \in \mathbb{F},\ f_1 \diamond_{\mathbb{F}} f_2 = \uparrow_\circ (f_1 \diamond_{\mathbb{R}} f_2) \tag{10}$$

For our needs, we also introduce the function $\downarrow_\circ\colon \mathbb{R} \to \mathbb{R}$ defined by $\downarrow_\circ (r) = r - \uparrow_\circ (r)$. To define the domain of error series, we assume that any line of code is identified by its number $\ell \in L$ and that $\mathcal{L}$ denotes the set $L \cup \{h\}$ where h is a special symbol used to denote the higher-order errors and which corresponds to no particular line of code. An error series r is defined by:

$$r = f\vec{\varepsilon}_\nu + \sum_{\ell\in\mathcal{L}} \omega^\ell \vec{\varepsilon}_\ell \tag{11}$$

In Equation (11), f is the floating-point number used by the processor. f is always attached to the formal variable $\vec{\varepsilon}_\nu$. A term $\omega^\ell \vec{\varepsilon}_\ell$ denotes the contribution to the global error of the first-order error introduced by the operation of line ℓ during the evaluation of r. $\omega^\ell \in \mathbb{R}$ is the scalar value of this error term and $\vec{\varepsilon}_\ell$ is a formal variable.

The elementary operations on error series are defined in Figure 3 for $r_1 = f_1\vec{\varepsilon}_\nu + \sum_{\ell\in\mathcal{L}^+} \omega_1^\ell \vec{\varepsilon}_\ell$ and $r_2 = f_2\vec{\varepsilon}_\nu + \sum_{\ell\in\mathcal{L}^+} \omega_2^\ell \vec{\varepsilon}_\ell$. ω_ν and f are used interchangeably to denote the coefficient of a variable $\vec{\varepsilon}_\nu$. The formal series $\sum_{\ell\in\mathcal{L}} \omega^\ell \vec{\varepsilon}_\ell$ related to the result of the operation $\diamond$ of Line ℓ_i contains the combination of the errors on the operands plus a new error term $\downarrow_\circ (f_1 \diamond_{\mathbb{R}} f_2)\vec{\varepsilon}_{\ell_i}$ corresponding to the error introduced by the operation of Line ℓ_i. The rules for addition and subtraction are natural. The elementary errors are added or subtracted componentwise in the series and the new error due to Line ℓ_i corresponds to the roundoff of the result. Multiplication requires more care because it introduces higher-order errors due to the multiplication of the first-order errors. The initial first-order errors $\omega_1^{\ell_1}\vec{\varepsilon}_{\ell_1}$ and $\omega_2^{\ell_2}\vec{\varepsilon}_{\ell_2}$ are multiplied by f_2 and f_1 respectively and products of first order error terms yield higher order error terms. $\otimes$ is a rewriting rule over $\mathcal{L}\cup\{\nu\}$ defined for $\ell, \ell' \in L$ by $\nu\otimes\nu = \nu$, $\nu \otimes \ell = \ell \otimes \nu = \ell$, $\ell \otimes \ell' = h$ and $\ell \otimes h = h \otimes \ell = h \otimes h = h$. Finally, the multiplication introduces a new first-order error $\downarrow_\circ (f_1 f_2)$ which is attached to the formal variable $\vec{\varepsilon}_{\ell_i}$ in order to indicate that this error is due to the product occurring at Line ℓ_i. The TMS320 assembler does not offer division but the semantics of this operation is given in [9].

Our semantics details the contribution to the global error of the first-order error terms and globally computes the higher-order error arising during the calculation. In practice, higher-order errors are often negligible. Our semantics allows us to determine the sources of imprecision while checking that the higher-order errors are actually negligible.

$$r_1 +^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 + f_2)\vec{\varepsilon} + \sum_{\ell \in \mathcal{L}} (\omega_1^\ell + \omega_2^\ell)\vec{\varepsilon}_\ell + \downarrow_\circ (f_1 + f_2)\vec{\varepsilon}_{\ell_i} \tag{12}$$

$$r_1 -^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 - f_2)\vec{\varepsilon} + \sum_{\ell \in \mathcal{L}} (\omega_1^\ell - \omega_2^\ell)\vec{\varepsilon}_\ell + \downarrow_\circ (f_1 - f_2)\vec{\varepsilon}_{\ell_i} \tag{13}$$

$$r_1 \times^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 f_2)\vec{\varepsilon} + \sum_{\ell_1, \ell_2 \in \mathcal{L} \cup \{\nu\},\ \ell_1 \otimes \ell_2 \neq \nu} \omega_1^{\ell_1} \omega_2^{\ell_2} \vec{\varepsilon}_{\ell_1 \otimes \ell_2} + \downarrow_\circ (f_1 f_2)\vec{\varepsilon}_{\ell_i} \tag{14}$$

Figure 3: Elementary operations over error series.

4. THE ANALYZER

The analyzer, written in OCaml, accepts assembler programs for the TMS320 C3X [14], a widely used DSP. It is based on the abstract interpretation of Section 2 and uses, for the floating-point computations, abstract error series in which the coefficients f and ω^ℓ respectively are abstracted by intervals of floating-point numbers and intervals of multiple precision numbers provided by the MPFR library [8].

The tool detects the runtime errors mentioned in Section 2 but the interface is mainly designed to validate and to help to improve the numerical accuracy of floating-point computations. In addition to the errors introduced by the elementary operations, error terms are introduced by the operations that copy values from 40 bits registers to 32 memory locations (`STF` and `PUSHF`). Next, the user may write assertions to specify the range and error of values stored in some registers or memory locations. For example the assertions `.float x [0.0,1.0,0.05,0.1]` states that the memory location `x` is initialized with a value belonging to $[0.0, 1.0]$ and that the error attached to this value belongs to $[0.05, 0.1]$. Once a program has been analyzed, the tool opens three windows, as shown in Figure 4. The bottom left window displays a synthesis of the results of the analysis. A colored box is drawn at the beginning of each line of code to indicate the intensity of the most important numerical inaccuracy at this line. More precisely, the analyzer finds the most inaccurate value in the abstract memory state of the current line and draws an orange box of proportional intensity. A low intensity corresponds to a small error (an invisible white box corresponds to no error at all) while large errors are represented by dark boxes. This enables the user to easily locate the inaccuracies. The rightmost window of Figure 4 displays the abstract value of a register or memory location at a given line of code. For values of type float, a range is given for the floating-point number as well as for the global error, i.e. the sum of the error terms of the series. Next, the user may see the source of this global error by opening a new window like the top left window of Figure 4. This window represents by an histogram the error series related to the considered floating-point value. In front of each line of code ℓ of the program, a bar proportional to the error term ω^ℓ of the series is displayed. The user can easily recognize which error terms contribute most to the global error: they correspond to the largest bars of the histogram. So, the programmer knows which statements need to be modified to improve the accuracy of the calculation. For example, if the precision loss due to the copy of a value from a register to memory is the source of a large inaccuracy, the programmer may decide to keep the value in the register.

The program analyzed in Figure 4 implements a first order recursive filter $y_n = ax_n + by_{n-1}$. The coefficients a and b are set to $[0.4, 0.45, 0.0, 0.005]$ and $[0.2, 0.3, 0.0, 0.01]$ in order to validate a family of filters with slightly inaccurate coefficients. x_n and y_{n-1} are stored in a two element array `x`. An assertion states that the entry signal x_n is bound by $0.5 \leq x_n \leq 1.0$ for all n. y_n is computed repeatedly, in an infinite loop. The rightmost and the top left windows display information on the output y, stored in `x[1]`. We can see in the rightmost window that the output always belongs to $[0.0, 0.6428571367]$ and that the error attached to this output never exceeds 0.016563320767. The error is never much more important than the initial error on the coefficients, independently of how long we iterate. The top left window gives the sources of this error. As expected in this example, the main errors are introduced by the multiplications whose operands are inaccurate and the second product introduces more inaccuracy than the first one.

5. CONCLUSION

In this article, we have introduced a static analyzer to detect runtime errors in DSP programs and to analyze the numerical accuracy of floating-point calculations. This analyzer is intended to precisely analyze industrial programs, mainly embedded airplane systems. For large codes, the histograms displayed by the tool are of primary interest to identify the sources of inaccuracies. The analysis of floating-point computations at the assembler level presents many advantages, with respect to higher-level languages. Firstly, the accuracy of a computation depends on the way it is compiled and, secondly, assembler programs contain additional information such as which values are stored in registers and consequently are represented with more bits than in memory.

Our toy language is representative of most existing assemblers but should be adapted for some processors. The main modification may concern the stack. In this article, we assumed that the stack was accessed by `PUSH` and `POP` mnemonics, as for the TMS320 processor [14]. Some assemblers such as the Sparc or 680X0 processors do not offer these operations and let the programmer directly manage a stack pointer SP. Our abstract stack should be revised in this case. A solution would be to assign an interval of integers to SP and to use a graph similar to the one of Section 2 in which the nodes would correspond to the concrete integer values taken by SP. Finally, we used separate compare and branch mnemonics as in most processors. However some processors have compare-and-branch mnemonics that merge the `CMPI`/`CMPF` and the `BLT`/`BEQ` statements. In this case, the analysis becomes simpler and the $\mathcal{E}$ environments introduced in Section 2 may only assign one memory state per line.

In further work, we plan to improve the iteration strategy

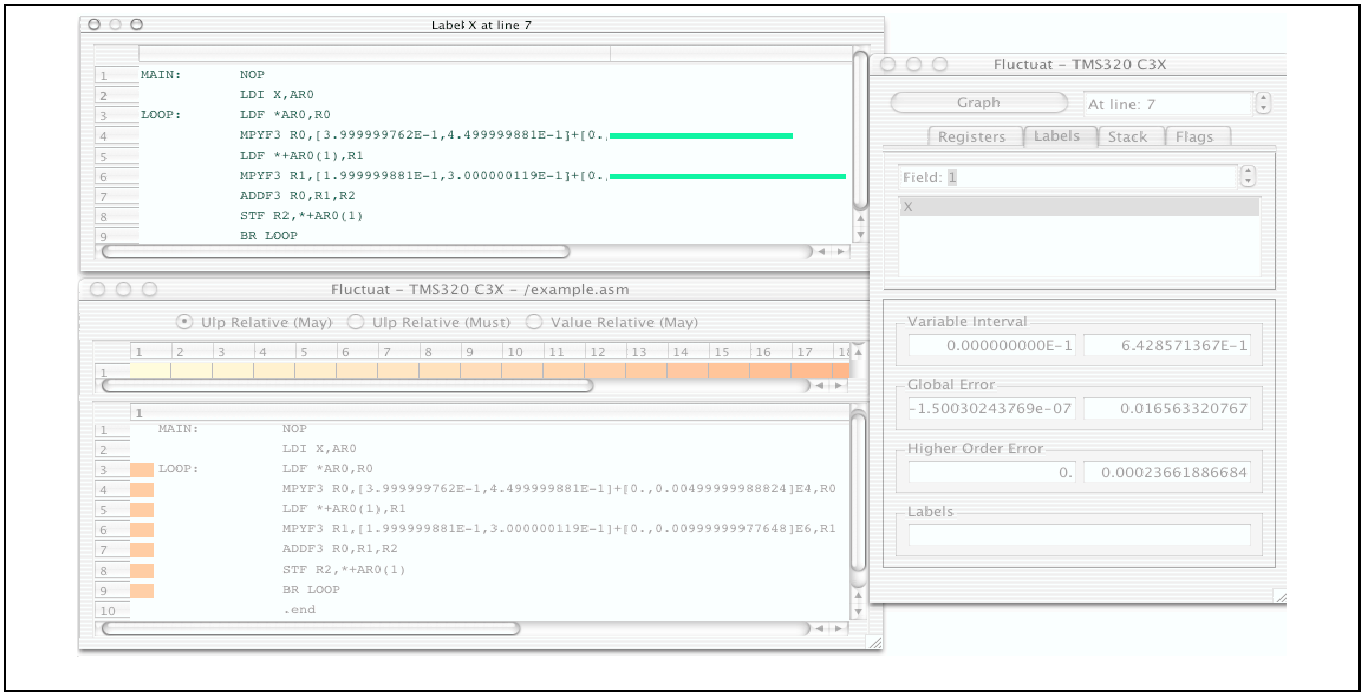

Figure 4: Screenshot of the analyzer.

of the analyzer by using dynamic partitionning techniques [11]. This would enable us to obtain more precise results in a minimal number of iterations instead of using analysis parameters like a widening threshold [11]. We also plan to add relational numerical domains for loops [10].

6. REFERENCES

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-1985 edition, 1985.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation, PLDI'03*. ACM Press, 2003.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages, POPL'77*, pages 238–252, 1977.

[4] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4):511–547, 1992.

[5] 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 *Embedded Software, EMSOFT'2001*, number 2211 in LNCS. Springer-Verlag, 2001.

[6] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[7] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in LNCS, pages 234–259. Springer-Verlag, 2001.

[8] G. Hanrot, V. Lefevre, Rouillier F., and P. Zimmermann. The MPFR library. Institut de Recherche en Informatique et Automatique, 2001.

[9] M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in LNCS, pages 194–208. Springer-Verlag, 2002.

[10] M. Martel. Static analysis of the numerical stability of loops. In *Static Analysis Symposium, SAS'02*, number 2477 in LNCS. Springer-Verlag, 2002.

[11] M. Martel. Improving the static analysis of loops by dynamic partitioning techniques. In *Source Code Analysis and Manipulation*. IEEE Press, 2003.

[12] S. Putot, E. Goubault, and M. Martel. Static analysis based validation of floating-point computations. In *Numerical Software with Result Verification*, number 2991 in LNCS, pages 306–313, 2004.

[13] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Principles of Programming Languages, POPL*, ACM Press, 2004.

[14] Texas Instruments. *TMS320C3x User's Guide*, 1997.

[15] H. Theiling. Ilp-based interprocedural path analysis. In *Embedded Software, EMSOFT'2001*, number 2491 in LNCS. Springer-Verlag, 2001.

[16] Z. Xu, T. Reps, and Miller B. Safety checking of machine code. In *Programming Language Design and Implementation, PLDI*. ACM, 2000.

[17] Z. Xu, T. Reps, and Miller B. Typestate checking of machine code. In *European Symposium on Programming, ESOP'01*, number 2028 in LNCS. Springer-Verlag, 2001.