PROPAGATION DE LA SENSIBILITÉ AUX ERREURS NUMÉRIQUES

DÉFINITION SENSIBILITÉ A L'ORIGINE DES ERREURS

- **Lorsqu'une erreur est introduite**

$$x = 1.0 / 10.0$$

l'erreur (flottant - réel) peut varier entre $[ulp_+(x)/2.0, ulp_-(x)/2.0]$.
Dans l'esprit du développeur, elle varie « librement » dans cet intervalle

erreur « moyenne » = $ulp_+(|x|)/4.0$, varie dans les intervalles $[0, ulp_+(|x|)/4.0]$ et $[-ulp_+(|x|)/4.0, 0]$ avec une « probabilité » identique ces intervalles

- **Définition d'une variable $t_{/ulp} \in [-1.0, +1.0]$**
  telle que $err_{/ulp} = +- (ulp_+(|x|) / (4.0)) + t_{/ulp} * ulp_+(|x|) / (4.0)$

$\delta(err_{/ulp})/\delta(t_{/ulp}) = ulp_+(|x|)/4.0$

valeur de cette dérivée partielle est de l'ordre de l'erreur moyenne

PROPAGATION SENSIBILITÉ A L'ORIGINE DES ERREURS

- **Lorsqu'une erreur est propagée**

$$z = x + y$$

$$err(z) = err(x) + err(y) + err_{+ulp}(z)$$

$\forall$ i expr ayant introduit une erreur, $\delta(err(z))/\delta(t_i) = \delta(err(x))/\delta(t_i) + \delta(err(y))/\delta(t_i)$
$\delta(err(z))/\delta(t_{+ulp}) = ulp_+(|z|)/4.0$

- **De manière générique, si z = op(x, y)**

$\forall$ i expr ayant introduit une erreur,

$$\delta(err(z))/\delta(t_i) = \delta(err(x))/\delta(t_i) * (\delta(op(x, y)/\delta x)) + \delta(err(y))/\delta(t_i) * (\delta(op(x, y)/\delta y))$$

$\delta(err(z))/\delta(t_{+ulp}) = ulp_+(|z|)/4.0$

PROBLÉMATIQUE

- **Quelles sont les instructions qui contribuent majoritairement à l'erreur finale ?**

- **Différentiation automatique de code**
  - Propagation forward
    - Mémoire shadow x 50 pour les front-ends ? Gérer les approximations
  - Propagation backward (ou lazy)
    - Mémoire shadow = std::shared_ptr, allocation dynamique ou position stream
    - Simplifications reportées sur les opérations binaires reportées
  - À la demande du δ-debug ?

- **Calcul matrice Jacobienne**
  - Nombreuses instructions flottantes, sans compter les boucles

- **Dérivées partielles en un point**
- **ou formule de dérivées partielles pour des ranges de valeurs**

IMPLÉMENTATION BACK-END ORIGINS

- **Propagation forward uniquement**
- **Implémentation de 2kloc, voire 1.5kloc**
- **Mémoire additionnelle :**

```
struct Origin {
   uint64_t symbol_id; // symbole créé dynamiquement, 0 si plusieurs symboles
   int operation_id;       // identificateur d'instruction défini statiquement
   float coefficient;      // valeur de la dérivée partielle par rapport au symbol_id
};

template <typename Type, int N>
struct OriginVector : public BaseOriginVector {
   Type value;
   std::array<Origin, N> origins;
   float coeff_without_origin = 0.0;
   int origins_size = 0;
};
```

DÉMONSTRATION SUR SQRT.C

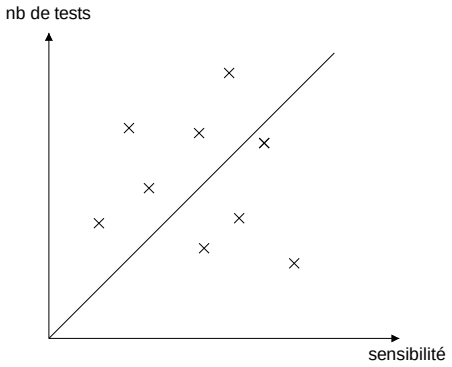
- **Utilisation front-end clang-instrument pour l'instrumentation**
- **TODOs**
  - Ne conserver qu'une identification par instruction
    - Faible probabilité de partage d'erreur entre expressions sur la pile et si c'est le cas, les résultats restent valides, ils sont juste sur-approximés

      double y = (x = f(…)) * x;

      double y = (x = f(…)), x * x;

      double y = (x = f(…) >= …) ? A / x * x : 0;
    - Optimise le remplissage du std::array
  - clang-instrument doit produire un dictionnaire des opérations
  - Pour l'analyse backward, un fichier de définition peut aussi être produit par l'analyse.
  - Rapport en Jupyter Notebook ?

1 couleur pour chacune des N instructions prépondérantes
x = résultat est sensible à …% w.r.t instruction i
dans …% des tests



FRONTS-ENDS C/C++

- **Avantages**
  - pas de limitation sur la taille de la shadow
  - compilations avec -g, -O0 permet de déboguer le code et de mettre des points d'arrêt relatifs à l'analyse en cours

- **Inconvénients**
  - Besoin de recompiler
  - Non robuste à certaines allocations/manipulation mémoires.
  - Gestion de headers communs entre code instrumenté et non-instrumenté
  - Link avec un code non-instrumenté
  - Comment assurer le suivi dans du code non-instrumenté
  - Back-end potentiellement dégradé pour code C par rapport au code C++
    - Utilisation de destructeurs dans le back-end
    - C ≠ C++ notamment les mots-clés sont différents: class peut être un nom de variable en C

- **Les codes flottants à repenser pour être instrumentables ?**
  - Un front-end de compilateur qui teste cette fonctionnalité ?

DIFFÉRENTS FRONT-END C/C++ POUR INSTRUMENTATION SANS MÉMOIRE SHADOW

- **Sans modification du code source (compiler: #define float …)**
  - gcc -include …/header.h -DACTIVATE_INSTRUMENTATION … -l…
  - clang include …/header.h -fplugin=…plugin.so \
    -Xclang -plugin-… -Xclang -option-name -Xclang -plugin-… \
    -Xclang $(patsubst %.c,%,$<) … -l…
- **Avec modification du code source (transpiler)**
  - cadnaizer et sa nouvelle version (cf LIP6), shaman (cf CEA)
  - instrumentation identique à celle du plugin clang
- **Pour un moteur d'analyse « formelle » (interpreter, compiler, transpiler)**
  - Ajout d'opérations de fusion (instructions ϕ forme SSA) requiert d'ajouter des opérations de type merge
  - Détection + qualification des tests instables

if (x < y) … transformé en …

memory_backup(…), memory_merge;
while (!finished) { **if** (x < y) …; memory_merge.store(…); }
memory_merge.restore(…)

- **Besoin d'un fichier additionnel pour délimiter l'instrumentation**

MÉMOIRE SHADOW ET UNISIM FULL SYSTEM

- **UNISIM ARM v8**
  - Simulateur host + virtio d'un code guest
  - Mémoire shadow au niveau host
  - Temps d'exécution x 100

- **Preuve de concept sur application Neural Network**
  - Boot de Linux + exécution du code
  - L'image d'entrée est teintée bit → bit de teinte
  - Propagation de la teinte via disques durs
  - sortie du programme

    [(7, 0.997771), (8, 0.0016825154), (86, 0.0003952935)]
    7, 8 et 86 sont les identifiants des classes à distinguer
    les teintes ne concernent que les scores de classes, soit:

    [(7, 0.**997771**), (8, 0.00**16825154**), (86, 0.000**3952935**)]
    printf produit les premiers « 0 » avec des dépendances de contrôle
    et les chiffres significatifs avec des dépendances de données (teintées)

EXPÉRIMENTATIONS À DÉCIDER

- **Partage d'expérimentations**

- **Problème de passage à l'échelle de ces techniques**
  - Définir des challenges pour savoir où elles sont intéressantes
  - Complémentarité avec le δ-debug

- **Complémentarité avec les autres analyses**
  - Choisir une étude et faire une preuve de concept dessus

TRAVAUX FUTURS

- **Mémoire shadow**
  - Augmenter la taille de la shadow d'Unisim avec une hiérarchie de mémoires shadow
  - Capacité à intégrer la shadow d'E-ACSL dans Unisim et la shadow d'Unisim dans E-ACSL
  - Capacité à intégrer la shadow de NSaN/Interflop dans Unisim ?

- **Analyses formelles et symboliques**
  - Nouveau back-end formel mélangeant abstrait et symbolique + heuristiques
  - Produire des résumés de code de bibliothèques non-instrumentables et utilisables par Verificarlo

- **Interface Utilisateur**
  - Interface orientée Debugger
  - Capacité à « attacher » un process provenant d'une autre analyse Interflop
  - Capacité à produire des résumés en « élargissant » les données provenant des tests

CONCLUSION

- **Un back-end pour propager les dérivées partielles par rapport aux erreurs introduites sur une instruction**
- **3 APIs différentes pour les back-ends contenant une mémoire additionnelle et sans shadow (destructeur, merge), en lien avec les différentes catégories de front-end**
- **De nombreux front-ends C/C++**
- **Des expérimentations partenariales à mener avec des challenges à poser**
- **Investissement dans l'analyse formelle (abstraite + symbolique) dirigée via une interface de debugger pour produire des « résumés » exploitables sur des portions de codes de calculs**