MINOTAuR : a timing-predictable RISC-V core featuring speculative execution

Alban Gruin

29 septembre 2023

IRIT, Univ. Toulouse III Paul Sabatier, CNRS

Contexte

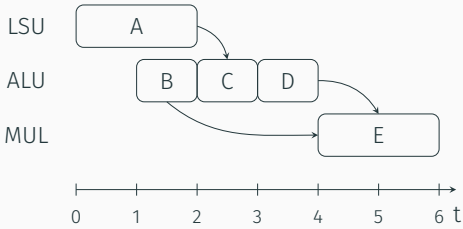
Les processeurs embarqués sont de plus en plus puissants et complexes

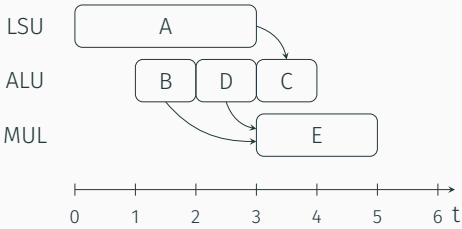
- Processeurs superscalaires, OoO, multicœurs…

Les interférences impactent les temps d'exécution

- Solutions ?
  1. Analyse WCET exhaustive
     - Pour chaque ressource partagée, considérer toutes les latences possibles
     - Généralement impossible (explosion du nombre d'états)
  2. Approche compositionnelle
     - Les latences sont analysées séparément, puis combinées
     - Nécessite l'absence d'*anomalies temporelles*

Anomalies temporelles i





Le pire cas local (latence la plus élevée pour une instruction) n'amène pas forcément au pire cas global (pire temps d'exécution de la séquence).

Anomalies temporelles : un faux problème ?

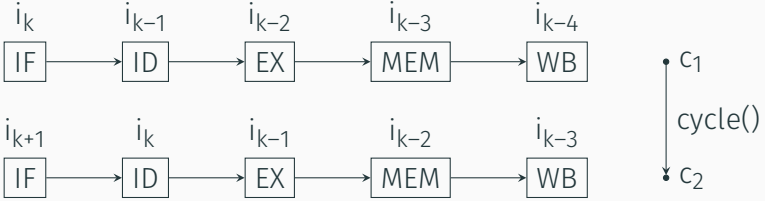
- Cas pathologiques trouvés sur certains modèles précis
- Impossible de démontrer l'absence d'anomalies pour la plupart des processeurs
- Hahn et Reineke ont présenté en 2018 un framework formel pour prouver des propriétés temporelles

Modèles de processeurs

Considérons une séquence d'instructions « abstraites » : $\mathcal{I} = i_0, i_1, ..., i_n$.

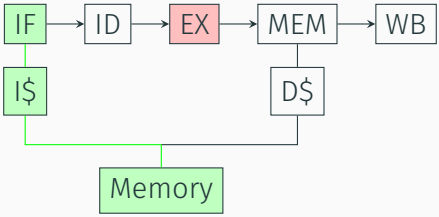
Progrès d'une instruction i_k : $(\mathrm{IF}, 1)$, état du pipeline $(c_1, c_2, ...)$.

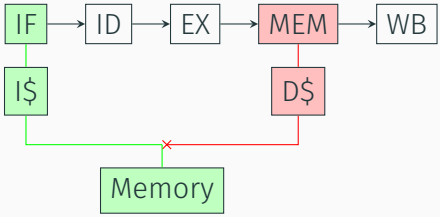
Fonction cycle(), pour passer d'un état à un autre.



Modifications sur le processeur

- Priorité aux instructions les plus anciennes sur les ressources partagées : **monotonicité**
- Pas d'accès au cache d'instructions si il y a une instr. mémoire dans le pipeline, pas de spéculation



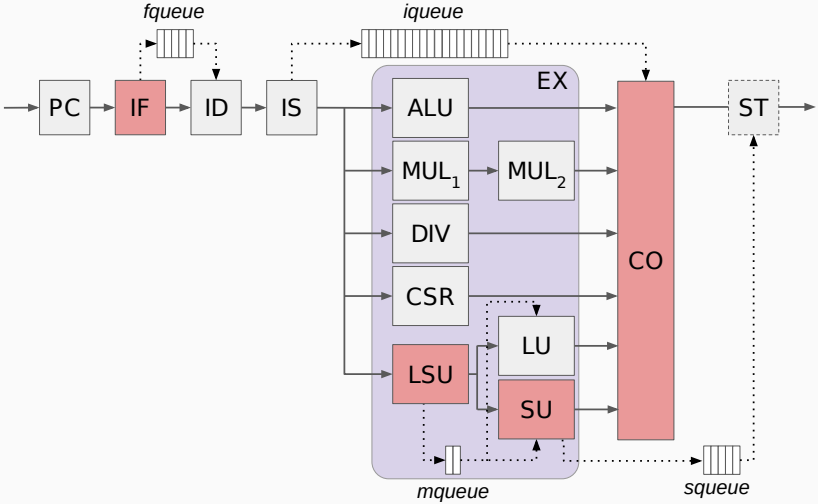


Objectif

Se baser sur l'existant

- Repartir de 0 : travail de titan
- Concevoir une architecture spécifique : besoin de changer les logiciels embarqués, voire de faire ses propres chaînes d'outil
- Modifier Ariane/CVA6 : utilisable avec `riscv-gcc` !

Présentation du processeur Ariane/CVA6/MINOTAuR



Extrait du modèle de MINOTAuR

$$
\begin{aligned}
&c.ready(i) := (c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i)) \\
&\qquad \vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i), i)) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{pc} \Rightarrow (ichit(i) \\
&\qquad\qquad \vee (\neg c.pending(i, branch) \wedge \neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{is} \Rightarrow (opc(i) \notin \{load, store, atomic\} \Rightarrow \neg c.pending(i, csr)) \\
&\qquad\qquad \wedge (opc(i) \in \{mul, div\} \Rightarrow \neg c.pending(i, div)) \\
&\qquad\qquad \wedge (\forall j < i \,.\, dep(i, j) \Rightarrow c.stg(j) \sqsupseteq_{\mathcal{S}} \textsc{co})) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{lsu} \Rightarrow (opc(i) \in \{store, atomic\} \wedge \neg c.pending(i, atomic)) \\
&\qquad\qquad \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic))))
\end{aligned}
$$

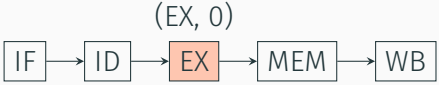
$$
\begin{aligned}
&c.free(s) := s \in \{\textsc{alu}, \textsc{mul}_1, \textsc{csr}, \textsc{mul}_2, \textsc{co}, post\} \\
&\qquad \vee (s \in \{\textsc{if}, \textsc{is}, \textsc{lsu}, \textsc{su}\} \wedge c.slot(s)) \\
&\qquad \vee (s \in \{\textsc{pc}, \textsc{id}, \textsc{div}, \textsc{lu}, \textsc{st}\} \wedge ((\neg \exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\
&\qquad \vee (\exists i \,.\, c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch))
\end{aligned}
$$

Prouver l'absence d'anomalies temporelles

- Assez facile quand i est dans deux étages différents, ou a deux latences différentes : peut rattraper, mais pas dépasser

État de i dans c_a (à gauche) et c_b (à droite) quand ils sont dans deux étages différents

- Plus compliqué quand $c_a(i) = c_b(i)$: il faut se baser sur des détails de la définition du pipeline.



État de i dans c_a et c_b

Impact sur les performances i

- Synthèse sur une carte Zybo Z7-20 avec Vivado 2021.1
- CoreMark et TACLe, compilé avec `gcc 10.2.0`
  - CoreMark compilé avec `-O3`
  - TACLe compilé avec `-O2`

Impact sur les performances ii

Version	LUTs	CoreMark	Moy. arith.	*Overhead* global
Ariane⁺	17 106	110,36		
MINOTAuR	17 176	110,58		
2 backups/RAS-16	21 782	108,78	0,39%	0,66%
4 backups/RAS-16	23 952	110,90	-1,28%	-2,00%
16 backups/RAS-2	19 439	110,89	-1,22%	-1,89%
16 backups/RAS-4	22 012	110,90	-1,31%	-2,05%

Jusqu'ici

Aspect théorique

- Prouvé l'absence d'anomalies temporelles pour MINOTAuR (à la main)
- Démontré un mécanisme pour éviter les anomalies temporelles lorsqu'un processeur a un *store buffer* (en Coq)
- *nb. Hahn et Reineke ont utilisé Z3 sur leur modèle, avec succès*

Aspect pratique

- MINOTAuR (dérivé du CV32A6) synthétisé sur un Zybo Z7-20
- Nouvelles modifs µArchi pour regagner de la performance
  - Plus de blocage MUL/ALU
  - Cache d'instructions LRU (vs. cache aléatoire, mieux pour le calcul du WCET)
- Description disponible en open-source

L'utilité de Coq

Les bons aspects de Coq

- Permet d'expliciter les hypothèses
- Donne un certain niveau d'assurance dans la preuve (pas d'oublis de cas, etc.)
- Coq peut être suffisament rapide (preuve du SIC en moins de 5 secondes)

Les mauvais aspects de Coq

- La preuve peut être longue à écrire
  - Preuve du SIC écrite en 1 semaine et demie, sans compter l'expérience nécessaire pour y arriver
  - Automatiser les preuves pour trouver un compromis rapide à écrire et prouver ?

Extrait du modèle de MINOTAuR (le retour)

$$
\begin{aligned}
&c.ready(i) := (c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i)) \\
&\qquad \vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i), i)) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{pc} \Rightarrow (ichit(i) \\
&\qquad\qquad \vee (\neg c.pending(i, branch) \wedge \neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{is} \Rightarrow (opc(i) \notin \{load, store, atomic\} \Rightarrow \neg c.pending(i, csr)) \\
&\qquad\qquad \wedge (opc(i) \in \{mul, div\} \Rightarrow \neg c.pending(i, div)) \\
&\qquad\qquad \wedge (\forall j < i \,.\, dep(i, j) \Rightarrow c.stg(j) \sqsupseteq_{\mathcal{S}} \textsc{co})) \\
&\qquad\quad \wedge (c.stg(i) = \textsc{lsu} \Rightarrow (opc(i) \in \{store, atomic\} \wedge \neg c.pending(i, atomic)) \\
&\qquad\qquad \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic))))
\end{aligned}
$$

$$
\begin{aligned}
&c.free(s) := s \in \{\textsc{alu}, \textsc{mul}_1, \textsc{csr}, \textsc{mul}_2, \textsc{co}, post\} \\
&\qquad \vee (s \in \{\textsc{if}, \textsc{is}, \textsc{lsu}, \textsc{su}\} \wedge c.slot(s)) \\
&\qquad \vee (s \in \{\textsc{pc}, \textsc{id}, \textsc{div}, \textsc{lu}, \textsc{st}\} \wedge ((\neg \exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\
&\qquad \vee (\exists i \,.\, c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch))
\end{aligned}
$$

Problèmes liés aux modèles temporels

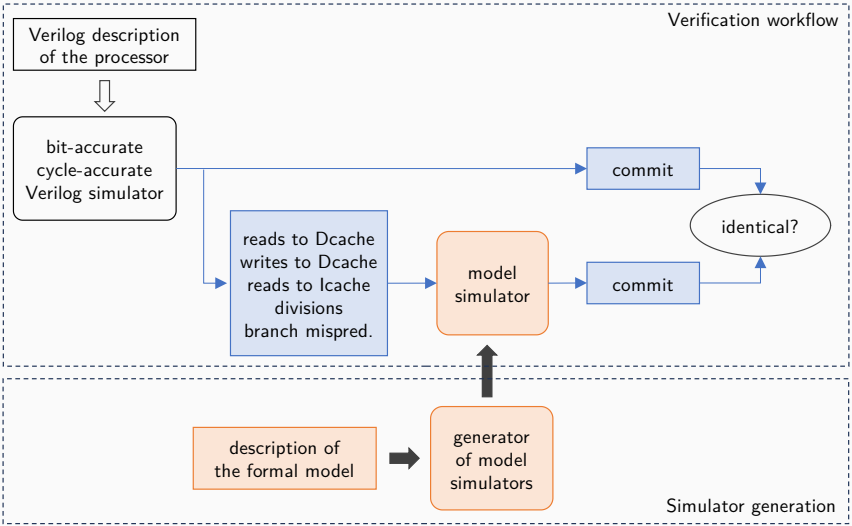
- Le modèle peut être difficile à écrire, et ne pas représenter correctement le processeur
- Les descriptions de matériel sont complexes
  - MINOTAuR $\rightarrow$ ~75 000 lignes de SystemVerilog...
- Les *datasheet* ne sont pas forcément précises
- Le processus n'est pas robuste

Comment valider un modèle temporel

Une méthodologie basée sur la simulation

- Infrastructure de simulation pour les modèles temporels
- Langage de description de modèles
- Compilateur de simulateur

Workflow de vérification



Traces

Type de trace	Champs
Icache	Adresses, opcodes, timings, annulations d'accès
Lectures dcache	Timings et annulations
Écritures dcache, divisions	Timings
Mispredict.	N° de cycle où il y a un mispredict
Commit	Adresse, n° de cycle de commit

Évaluation sur MINOTAuR

- 2 jours pour générer l'ensemble des traces avec Questa
- Validation du modèle en ~1h
- Plusieurs erreurs ont été trouvées, corrigées, puis revalidées.
  - Gestion des dépendances de données.
  - Stall de la LU après un miss.
  - Dispatch des instructions arith. après un CSR.
- Après correction, le simulateur génère les mêmes traces que le processeur.

Génération du simulateur

- Passer de la logique des prédicats à un langage rapide (ex. C++) est aussi une source d'erreur.
- Utilisation d'un langage spécifique pour assister cette traduction.

```
let ready(opc, limit c, i, pwrong) =
  (stg(c, i) <> Pre /\ !pending(opc, c, i, Branch) /\ pwrong)
  \/ (cnt(c, i) = 0 /\ isnext(c, stg(c, i), i) /\
      (stg(c, i) = IS ->
        (opc[i] in {Mul, Div} -> !pending(opc, c, i, Div))
        /\ (forall j in c, (j < i -> !dep(opc, c, i, j))))
     /\ (stg(c, i) = LSU ->
         (opc[i] in {Store, Atomic} /\ !pending(opc, c, i, Atomic))
         \/ (opc[i] = Load /\ !pending(opc, c, i, Atomic))))
```

Conclusion

- Nous avons modifié le processeur Ariane/CVA6 pour le rendre prédictible
  - Impact sur les performances : moins de 5 %.
- Modélisation formelle du processeur, démonstration de la propriété de monotonicité
  - Pour l'instant à la main (fastidieux)
  - Coq : en cours
- Méthodologie de validation du modèle formel, basée sur de la simulation de traces (générées avec Questa pour l'instant)

Merci !