TP OTAWA ETR 2021

Thomas Carle, Hugues Cassé

OTAWA est un ensemble de bibliothèques open-source d'analyse statique sous licence LGPL. Ces bibliothèques ciblent particulièrement l'analyse de programmes au niveau binaire, et le calcul de pire-temps d'exécution (WCET).

Pour le TP, OTAWA a été installé dans une machine virtuelle, dans le dossier ~/OTAWA . Les exécutables otawa sont disponibles dans ~/OTAWA/bin et ce dossier a été ajouté dans le PATH de la machine virtuelle.
Les sources pour le TP sont disponibles dans le dossier ~/otawa_tp .

Dans ce TP, nous allons prendre en main les différentes commandes fournies de base avec OTAWA, afin d'analyser le pire temps d'exécution de programmes, et d'obtenir des statistiques sur ces programmes.

I. Prise en main d'OTAWA

Répertoire : ~/otawa_tp/bs
Source : bs.c
Le benchmark bs est une petite application composée de 2 fonctions, **main** et **binary_search** qui réalise une recherche binaire dans un tableau. Il n'y a qu'une boucle dans ce programme (présente dans **binary_search**), qui doit nécessairement être bornée si on souhaite pouvoir obtenir un WCET.
Cette petite appli est l'occasion pour nous d'introduire le process allant du code source jusqu'à l'analyse de WCET en passant par la compilation pour une architecture bare-metal et par l'obtention de bornes de boucles.

I.1. Calcul du WCET

1. Dans le dossier ~/otawa_tp/bs, compilez l'application avec la commande **make**. Dans cette commande, qui lance le compilateur gcc pour cibles ARM bare-metal, vous pouvez constater que nous utilisons les options suivantes :
    - -g3 : pour obtenir de l'information de debug, qui nous permet de faire le lien entre le code source et le binaire
    - -static : pour assurer que le binaire ne repose pas sur des bibliothèques partagées
    - -nostartfiles : le démarrage du programme est contenu dans le programme, pas besoin de Linux

2. Lancez le calcul de WCET pour la fonction main sur une architecture lpc2138 :

   ```
   $> owcet -s lpc2138 bs.elf
   ```

   lpc2138 correspond à la micro architecture d'une famille de microcontrôleurs ARM produits par NXP.

   La commande owcet doit vous retourner une erreur :

   ```
   INFO: plugged otawa/lpc2138 (.../lib/otawa/otawa/lpc2138.so)
   WARNING: otawa::util:: FlowFactLoader 1.4.0: no flow fact file
         for bs.elf
   WARNING: otawa::ipet:: FlowFactLoader 2.0.0: no limit for the loop
         at binary_search + 0xb4 (000101b4).
   WARNING: otawa::ipet:: FlowFactLoader 2.0.0: in the context
         [FUN (000101dc), CALL (000101 e8), FUN (00010100)]
   ERROR: otawa :: Weighter (1.0.0): cannot compute weight for loop
         at BB 2 (000101b4)
   ```

   Cette erreur nous indique qu'une boucle est présente dans le programme, mais qu'aucune information de borne n'a été fournie pour celle-ci. Dans ces conditions, le solveur ILP d'owcet, en cherchant à maximiser le WCET ne pourrait que trouver une solution infinie.

   Dans le détail, owcet nous indique que la boucle en question est présente dans le binaire à l'adresse 0x101b4.

3. Afin de fournir l'information manquante, générez un fichier de **flow fact** avec la commande **mkff** :

   ```
   $> mkff bs.elf > bs.ff
   ```

   Il faut ensuite éditer à la main l'information de borne de boucle dans le fichier bs.ff . Dans la ligne correspondant à la boucle, on remplace le "?" par cette information. On utilise la syntaxe "max N", où N est le nombre d'itérations maximum de la boucle. Remarquez que le fichier de flow fact contient toutes les informations nécessaires pour retrouver la boucle correspondante dans le fichier source (et ainsi faciliter notre recherche de la valeur maximale) : la fonction contenant la boucle (binary_search), et la ligne dans le source (bs.c:88). Dans notre cas, on peut trouver que la borne maximale de la boucle est de 10 itérations.

4. On peut désormais relancer owcet, qui va prendre en compte le fichier de flow fact pour borner la boucle et ainsi limiter la maximisation dans le solveur ILP :

   ```
   $> owcet -s lpc2138 bs.elf
   WCET[main] = 1962 cycles
   ```

   (le nombre de cycles trouvé dépend de la version du compilateur que l'on utilise, et peut donc varier).

I.2. Détails sur le WCET

L'obtention du WCET est nécessaire à l'étude d'ordonnançabilité et à la validation temporelle d'un système temps-réel dur. Que faire si le système n'est pas ordonnançable ? On peut soit retravailler le système, par exemple en enlevant des tâches, en les re-découpant ou en changeant leurs périodes, soit retravailler le code des

tâches pour essayer de réduire leur WCET. OTAWA peut aider à travailler sur les tâches, en fournissant des informations détaillées sur les portions de code qui prennent le plus de temps à s'exécuter (ou qui en tout cas ont le plus fort impact sur le calcul du WCET). Pour obtenir ces détails, on demande à OTAWA de produire des statistiques sur WCET calculé.

1. Re-calculez le WCET, cette fois en générant des statistiques :

```
$> owcet -s lpc2138 bs.elf --stats -S
WCET[main] = 444 cycles
Total Execution Count: avg=4.5, max=11, min=0
Total Execution Time: total=1962, avg=196.2, max=820, min=0
```

   En plus du WCET, on obtient cette fois de l'information sur la durée des blocs de base composant le programme et sur le nombre de fois qu'ils sont exécutés dans le WCET :
   - Dans la configuration d'exécution menant au WCET, en moyenne chaque bloc est exécuté 4.5 fois, le bloc le plus exécuté l'est 11 fois et le moins exécuté, 0 fois
   - Dans la configuration d'exécution menant au WCET, en moyenne chaque bloc compte pour 196.2 cycles, le bloc le plus influent compte pour 820 cycles et le moins influent compte pour 0 cycle.
2. Pour obtenir plus de détails, par exemple les statistique par ligne de code source, on peut générer les sources décorées avec les statistiques:

```
$> otawa-stat.py -S main -a
```

   Où **main** est le nom de la fonction correspondant à la tâche analysée.
   On peut ensuite afficher les sources décorées :

```
$> xdg-open main-otawa/src/index.html
```

   - Quelles sont les lignes de source les plus coûteuses en temps d'exécution ?
   - Quelles sont les lignes les plus exécutées ?

   Pour obtenir des informations encore plus détaillées (mais plus difficilement exploitables pour réécrire le code des tâches), on peut obtenir les statistiques au niveau du CFG (assembleur) :

```
$> otawa-stat.py main -G -S -a
```

   On peut ensuite les afficher avec les commandes suivantes :

```
$> otawa-xdot.py main-otawa/ipet-total_count-cfg/index.dot &
$> otawa-xdot.py main-otawa/ipet-total_time-cfg/index.dot &
```

   Dans l'affichage d'otawa-xdot.py, vous pouvez naviguer entre les différentes fonctions en cliquant sur les blocs correspondant à leurs appels.
   - Quels blocs sont les plus coûteux en temps d'exécution, en considérant toutes les fonctions du programme ?
   - Quels sont les blocs les plus fréquemment exécutés dans la configuration d'exécution menant au WCET ?
   - Ces blocs correspondent-ils aux lignes de code source ?
   - Pouvez-vous déterminer le chemin d'exécution menant au WCET ?

I.3. Détail des calculs d'OTAWA

Dans cette section nous allons afficher la liste des analyses pratiquées par OTAWA afin d'arriver au WCET. Pour cela on ajoutera simplement l'option --log proc lors de l'appel de owcet :

```
$> owcet -s lpc2138 bs.elf --log proc
```

Chaque ligne de la sortie correspond à une analyse pratiquée par OTAWA.

```
RUNNING: otawa::dfa:: InitialStateBuilder (1.0.0)
RUNNING: otawa::util:: FlowFactLoader (1.4.0)
RUNNING: otawa:: LabelSetter (1.0.0)
RUNNING: otawa::view:: Maker (1.0.0)
RUNNING: otawa:: CFGCollector (2.1.0)
RUNNING: otawa:: LoopReductor (2.0.1)
RUNNING: otawa:: Virtualizer (2.0.0)
RUNNING: otawa:: MemoryProcessor (1.0.0)
RUNNING: otawa:: lpc2138 :: AbsMAMBlockBuilder (2.0.0)
RUNNING: otawa:: Dominance (1.2.0)
RUNNING: otawa:: LoopInfoBuilder (2.0.0)
RUNNING: otawa:: lpc2138 :: CATMAMBuilder (2.0.0)
RUNNING: otawa:: ProcessorProcessor (1.0.0)
RUNNING: otawa:: lpc2138 :: MAMEventBuilder (1.0.0)
RUNNING: otawa::ipet:: ILPSystemGetter (1.1.0)
RUNNING: otawa::ipet:: VarAssignment (1.0.0)
RUNNING: otawa:: ExtendedLoopBuilder (1.0.0)
RUNNING: otawa:: CFGChecker (1.0.0)
RUNNING: otawa::ipet:: FlowFactLoader (2.0.0)
RUNNING: otawa:: Weighter (1.0.0)
RUNNING: otawa:: etime:: StandardEventBuilder (1.0.0)
RUNNING: otawa:: lpc2138 :: BBTime (2.0.0)
RUNNING: otawa::ipet:: BasicConstraintsBuilder (1.0.0)
RUNNING: otawa::ipet:: FlowFactConstraintBuilder (1.1.0)
RUNNING: otawa::ipet:: WCETComputation (1.1.0)
```

Les analyses les plus courantes sont :

- util::FlowFactLoader : chargement des bornes de boucles
- CFGCollector : construction du CFG à partir du code binaire
- MemoryProcessor : chargement de la description de la mémoire
- LoopInfoBuilder : détection des boucles dans le CFG
- ProcessorProcessor : chargement de la configuration du processeur
- ipet::FlowFactLoader : assignation des bornes de boucles aux blocs de base
- ipet::BasicConstraintBuilder : construction des contraintes de flot de contrôle dans le système ILP

- ipet::FlowFactConstraintBuilder : construction des contraintes de bornes de boucles dans le système ILP
- ipet::WCETComputation : calcul du WCET

D'autres analyses sont spécifiques à la cible architecturale spécifiée :

- lpc2138::CATMAMBuilder : analyse des prefetch buffers de la mémoire flash (qui contient le programme)
- lpc2138::BBTime : calcul de la durée d'exécution de chaque bloc de base

II. Borner les boucles avec oRange

Répertoire : ~/otawa_tp/crc
Source : crc.c

Crc est une petite application qui réalise, comme son nom l'indique, des calculs de CRC. Le programme est cependant plus complexe que bs, et contient notamment plusieurs boucles. Dans un premier temps, vous allez déterminer les bornes de boucles à la main, puis dans un second temps, avec l'outil oRange, afin de voir la différence.

1. Compilez le fichier source pour obtenir l'exécutable
2. À l'aide de mkff, générez le fichier crc.ff
3. Remplissez les informations de bornes de boucles manquantes, en vous aidant du code source.
4. Calculez le WCET et générez les statistiques au niveau du CFG.
5. Affichez le CFG décoré avec les informations d'ipet-total_count et naviguez jusqu'à la fonction icrc. Localisez le bloc de base A correspondant à la ligne crc.c:93, qui représente la tête de boucle d'une boucle dont le corps est le bloc B de la ligne crc.c:94. Pourquoi le bloc A est-il exécuté une fois de plus que le bloc B ?
6. Dans le CFG, la fonction main contient 2 appels à icrc. Si vous cliquez sur les deux blocs d'appels, les CFGs correspondants ne sont pas les mêmes. Pourquoi ?
7. Fermez les fenêtres otawa-xdot et notez le WCET calculé (on l'appellera WCET initial dans la suite). Supprimez le fichier crc.ff (ou changez l'extension).
8. Lorsque les bornes de boucles sont compliquées à obtenir, on utilise l'outil oRange :

   ```
   $>orange crc.c main -o crc.ffx
   ```

   Cette commande crée un fichier au format XML nommé crc.ffx. Ouvrez ce fichier avec un éditeur de texte et observez comment les bornes de boucles sont fournies, en prenant en compte les chaînes d'appels de sous-programmes menant à une boucle en particulier. Désormais quelle est la borne de boucle à la ligne crc.c:93 ?

9. Calculez à nouveau le WCET, cette fois-ci avec le fichier ffx. Le nouveau WCET est inférieur au WCET initial. Pouvez-vous expliquer cette différence grâce aux statistiques et au fichier ffx généré par oRange ?

III. Bornes de boucles et utilisation du mot clé total

Répertoire : ~otawa_tp/bubble
Source : bubble.c
Bubble est une application de tri bulle sur un tableau.
Cet exercice illustre le fait que fournir uniquement une borne maximum du nombre d'itérations de boucle ne permet pas toujours d'obtenir un WCET précis, en particulier lorsque la borne d'une boucle dépend du contexte. On va travailler sur un cas pathologique dans lequel le calcul du nombre total d'itérations permet de réduire significativement le WCET.
On utilisera ainsi deux types de bornes :

- Borne max : le nombre maximum d'itérations qu'une boucle peut effectuer une fois qu'on entre dedans, peu importe le contexte
- Borne totale : le nombre total d'itérations réalisées par une boucle, tout au long du programme (y compris si on y entre plusieurs fois)

1. Compilez le programme.
2. Avec mkff, créez le fichier bubble.ff .
3. Remplissez les champs manquants de bubble.ff en utilisant le fichier source.
4. Calculez le WCET, avec les statistiques.
5. Générez la version CFG des statistiques.
6. A l'aide de ce CFG, notez le nombre d'itérations réalisées par le bloc de base servant de tête de boucle à la ligne bubble.c:15 .
7. En regardant le fichier source, calculez le nombre total d'itérations réalisées par cette boucle.
8. Qu'observez-vous ? Pourquoi ?
9. Pour résoudre le problème, complétez le fichier bubble.ff en utilisant pour la boucle après "max N", la syntaxe "total M", où M est le nombre total d'itérations de la boucle tout au long de l'exécution.
10. Re-calculez le WCET et comparez-le avec la version précédente.

IV. Application multi-tâches

Répertoire : ~otawa_tp/helico
Source : helico.c
helico.c est une application de contrôle d'un drone quadcopter. Elle mène une mission simple : décoller, maintenir la stabilité et atterrir. Cette application est composée de plusieurs tâches temps-réel s'exécutant en séquence dans une boucle infinie dans le main(), avec une fréquence de 1ms.

OTAWA ne peut donc pas calculer le WCET de cette application comme un tout, mais en analysant le contenu du main(), on constate qu'elle est composée de trois tâches :

- updateADC()
- action()
- doPWM()

Pour vérifier l'ordonnançabilité du système, on doit calculer séparément le WCET des fonctions qui implémentent ces 3 tâches.

1. Compilez l'application
2. Calculez le WCET de la fonction doPWM() avec la commande (cette fonction n'a pas de boucle) :
   ```
   $> owcet -s lpc2138 helico.elf doPWM
   ```
3. action() continent une boucle, il nous faut donc générer un fichier de flow fact :
   ```
   $> mkff helico.elf action > action.ff
   ```
   puis renseigner la borne de boucle.
4. On peut désormais calculer le WCET de la fonction action() :
   ```
   $> owcet -s lpc2138 helico.elf action -f action.ff
   ```
5. Calculez de même le WCET de updateADC() en utilisant oRange pour obtenir les bornes de boucle.
6. Si on considère que le code de gestion de la boucle infinie de la fonction main() prend moins de 50 cycles par tour de boucle, quel est le temps d'exécution complet d'un tour de boucle ?
7. En considérant que la période des tâches est de 1ms, est-il suffisant d'utiliser un processeur LPC2138 cadencé à 16MHz ? Calculez la fréquence minimale permettant d'exécuter la boucle avec une période de 1ms.

V. Améliorer le WCET

Répertoire : ~/otawa_tp/helico
Source : helico.c

Cet exercice poursuit le précédent. On va entrer dans le détail du code afin de réduire le WCET de la boucle infinie du main().

1. En considérant que la boucle infinie du main() a une période de 1ms, regardez dans le code source de la fonction doPWM() pour déterminer la période de la fonction updatePWM() qui réalise le "vrai" travail de doPWM().
2. Calculez le WCET des fonctions suivantes, en utilisant des flow facts générés par oRange :
   - doGyroChannel()
   - doAROMXChannel()
   - doAROMYChannel()
   - doAROMZChannel()

   Qu'observez-vous sur ces WCETs ?
3. En prenant en compte le fait que (a) à chaque itération de la boucle infinie du main(), updateADC() est appelée 4 fois avec les 4 valeurs possibles pour currentChannel et que (b) dans la fonction updateADC() une seule des fonctions

précédentes est appelée, pouvez-vous estimer le WCET des 4 appels d'updateADC() dans la sous-boucle du main() ?

4. En utilisant cette estimation, approximez le WCET total d'une itération de la boucle infinie du main. Y-a-t'il une différence avec le WCET de l'exercice précédent ?
5. Modifiez l'application helico pour réduire la surestimation du WCET observée dans la question précédente.
6. Re-calculez la fréquence minimale d'un processeur permettant de faire tourner cette application en respectant la contrainte temps-réel de 1 ms.

VI. Flot de contrôle complexe

Répertoire : ~/otawa_tp/control
Source : control.c

Dans cet exercice on va voir qu'il est possible de fournir des informations de flot de contrôle à OTAWA lorsque celui-ci est complexe.

1. Compilez le programme control
2. Essayez de calculer le WCET de la fonction main. Vous devriez obtenir l'erreur suivante :
   ```
   WARNING: otawa:: CFGChecker 1.0.0: CFG _exit is not connected
   (this may be due to infinite or unresolved branches ).
   ERROR: CFG checking has show anomalies (see above for details ).
   ```
3. Pour comprendre cette erreur, générez une représentation graphique du CFG de l'application :
   ```
   $> dumpcfg -Mds control.elf
   ```
   Pour l'afficher :
   ```
   $> otawa-xdot.py main-otawa/cfg/index.dot &
   ```
   On remarque deux choses :
   - L'appel par pointeur de fonction ligne 34 n'a pas été résolu par OTAWA
   - Le CFG de la fonction _exit() n'est pas connecté : cela vient de la dernière instruction de la fonction _exit() : SWI qui effectue un appel système à la fin du programme control.

   Nous devons aider OTAWA à dépasser ces difficultés.
4. D'abord, générez le fichier de flow facts control.ff avec mkff. Editez ce fichier. Il contient de nouvelles commandes qui adressent les problèmes précédents.
   a. La première commande représente l'appel de fonction par pointeur : multicall. Vous devez remplacer le '?' par le nom de la ou des fonctions qui peuvent être appelées par ce pointeur (en cas de plusieurs fonctions, séparer les noms par des virgules).
   b. La deuxième commande concerne l'instruction SWI de la fonction _exit(). mkff nous propose de considérer cette fonction comme une instruction

sans retour ou comme un appel à plusieurs fonctions. Retirez la ligne incorrecte dans le .ff.

5. Re-générez le CFG, et vérifiez qu'il est consistant, c'est à dire connecté et dans branchement ou appel non résolu.
6. Calculez le WCET de cette application.