RENPAR/CFSE/SYMPA
Paris, cit des sciences et de l'industrie, 24 – 27 avril 2001

Devil : un IDL pour les contrôleurs de périphériques

Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, Gilles Muller

Groupe Compose, IRISA/LaBRI, `http://compose.labri.fr/`

Résumé
Pour suivre la cadence effrénée à laquelle de nouveaux périphériques sortent sur le marché, les pilotes correspondants doivent souvent être développés dans l'urgence. Bien que les pilotes de périphériques soient des composants critiques des systèmes d'exploitation, leur processus de développement est rudimentaire. Cela est particulièrement flagrant en ce qui concerne la couche basse des pilotes, chargée de la communication directe avec les contrôleurs de périphériques. D'une part, développer cette couche nécessite l'étude de documentations trop souvent imprécises ou incomplètes. D'autre part, cette couche utilise surtout des opérateurs bit à bit, pour lesquels le langage C n'offre pas plus de sûreté que l'assembleur. Il en ressort que la couche basse des pilotes est difficile à écrire, et souvent source d'erreurs. Cet article présente une nouvelle approche du développement de la couche basse des pilotes. Cette approche repose sur Devil, un language de définition d'interfaces (IDL) dédié aux contrôleurs de périphériques. Cet IDL permet d'écrire une spécification de haut niveau de l'interface de programmation d'un contrôleur. La cohérence d'une spécification Devil est vérifiée automatiquement par un compilateur, lequel peut ensuite générer un code efficace pour la couche basse du pilote.

Mots-clés : Pilotes de périphériques, Langages dédiés, IDL

1. Introduction

Les pilotes de périphériques sont des composants clé d'un système d'exploitation. D'une part, ce sont eux qui rendent les innovations matérielles disponibles pour l'utilisateur final. Tout retard dans le développement d'un pilote peut donc remettre en cause la compétitivité du produit matériel. D'autre part, chaque pilote fait partie intégrante du système d'exploitation. Il s'exécute en mode noyau, et le moindre bogue peut donc compromettre la stabilité du système entier.

Malgré le rôle prépondérant des pilotes, leur processus de développement n'a que très peu évolué. C'est particulièrement préoccupant losqu'on considère la couche basse des pilotes, c'est-à-dire la partie qui communique directement avec le matériel. Ecrire cette couche est réputé être une tâche difficile, car fastidieuse et source d'erreur.
En effet, la couche basse des pilotes se programme à un niveau d'abstraction très bas, en utilisant principalement des opérateurs bit-à-bit (combinaisons logiques et décalages) ainsi que des accès explicites aux adresses d'entrée/sortie. Un tel niveau d'abstraction étant proche de l'assembleur, le code est difficile à lire et les erreurs sont difficiles à débusquer. De plus, le typage du langage C n'est d'aucun secours, car tous les opérandes sont des entiers. En fait, la communication avec le matériel est une application qui atteind les limites d'un langage généraliste tel que C.
En ce qui concerne la documentation du matériel, celle-ci est souvent entâchée par des imprécisions ou des erreurs typographiques. Chaque constructeur a son vocabulaire et sa façon de présenter les choses. Par conséquent, écrire la couche basse d'un pilote oblige d'abord à se lancer dans une quête laborieuse, menant à d'obscures incantations. Non seulement ces incantations n'ont pas toujours l'effet escompté, mais elles rendent le code difficile à réutiliser et à mettre à jour.

Notre approche

Cet article présente une nouvelle approche pour l'écriture de la couche basse des pilotes de périphériques. Notre idée est de modéliser cette couche dans un langage de haut niveau dédié à cette tâche. Cela présente deux avantages : une écriture plus guidée qu'avec le langage C, et la possibilité de vérifier automatiquement d'importantes propriétés de sûreté. Une fois la couche basse spécifiée, un compilateur peut en générer une implémentation.

Nous introduisons un language de définition d'interface (IDL), nommé Devil, qui permet de spécifier l'interface de programmation des contrôleurs de périphériques [9]. Les IDL sont déjà très utilisés dans les systèmes d'exploitation (SE) modernes, que ce soit dans les SE distribués pour masquer l'hétérogénéité et la complexité de la construction des messages [2, 10], ou dans les SE modulaires pour coller les composants entre eux [1, 6, 7]. À l'instar des IDL pour RPC qui définissent des opérations et leurs types d'entrée/sortie, Devil permet de spécifier l'interface fonctionnelle d'un contrôleur, ainsi que sa sémantique. Les abstractions et les constructions syntaxiques de Devil ont été conçues pour faciliter l'écriture de cette spécification, et la rendre aussi intuitive que possible. Une fois cette étape franchie, un compilateur utilise la spécification pour génèrer automatiquement la couche basse du pilote, sous la forme d'une bibliothèque conforme à l'interface fonctionnelle spécifiée. Cette bibliothèque encapsule le code C de bas niveau responsable de la communication directe avec le contrôleur. De plus, si le programmeur le souhaite, des outils permettent de vérifier certaines propriétés critiques de sûreté, que ce soit à la compilation ou à l'exécution.
L'emploi d'un IDL facilite souvent la réutilisation du code, et c'est le cas pour Devil. Une spécification Devil peut être réutilisée pour différentes plateformes : il suffit de disposer du compilateur adéquat. Dans notre vision, les spécifications Devil devraient être écrites pas les constructeurs des contrôleurs, ou du moins être disponibles publiquement sous la forme d'une bibliothèque en ligne. Le programmeur souhaitant écriture un nouveau pilote pour un contrôleur courant n'aurait alors pas besoin d'écrire une spécification Devil : il pourrait la prendre dans la bibliothèque et la compiler pour sa plateforme.

La suite de cet article s'organise ainsi. La section 2 présente le langage Devil. La section 3 décrit les propriétés de sûreté pouvant être vérifiées statiquement ou dynamiquement. La section 4 montre par des tests de performances que notre approche est viable en pratique. La section 6 parle des travaux connexes. La section 5 expose l'idée d'une bibliothèque en ligne de spécifications, en discutant de ce qui existe déjà et de ce qu'il reste à faire. Enfin, la section 7 conclut cet article.

2. Devil

Devil est un IDL dedié à la spécification de l'interface de programmation des contrôleurs de périphériques. Pour concevoir Devil, nous avons étudié les pilotes (principalement Linux) d'un large spectre de périphériques : cartes Ethernet, cartes graphiques, cartes son, souris, contrôleur DMA et contrôleur d'interruptions. Cette étude a été étayée par des informations recueillies dans la littérature [5, 14], dans la documentation des circuits, et en discutant avec des experts de la programmation de pilotes pour Windows, pour Linux, et pour divers systèmes enfouis.
Concrètement, un contrôleur de périphérique offre au processeur central une interface de programmation faisant appel à trois niveaux d'abstraction : *les ports*, *les registres*, et les *champs de bits*. Dans Devil, les champs de bits sont explicitement typés, ce qui fait d'eux des entités de plus haut niveau appelées *variables de contrôle*. Le point d'entrée d'une spécification Devil est la déclaration du contrôleur, paramétrée par des ports (ou des intervalles de ports) qui sont l'abstraction des adresses physiques donnant accès au contrôleur. À partir de ces ports sont déclarés les registres, qui définissent la granularité des interactions avec le contrôleur. Enfin, les variables de contrôle sont déclarées à partir des registres. Elles constituent l'interface fonctionnelle du contrôleur. Par conséquent, les variables sont les seules abstractions de Devil qui soient visibles depuis les couches supérieures du pilote. Concrètement, pour chaque variable `xx`, le compilateur Devil génère une

```
device logitech_busmouse (base : bit[8] port @ {0..3})                          1
{
   // Signature register (SR)
   register sig_reg = base @ 1 : bit[8];                                        4
   variable signature = sig_reg, volatile, write trigger : int(8);              5

   // Configuration register (CR)
   register cr = write base @ 3, mask '1001000.' : bit[8];                      8
   variable config = cr[0] : { CONFIGURATION => '1', DEFAULT_MODE => '0' };     9

   // Interrupt register
   register interrupt_reg = write base @ 2, mask '000.0000' : bit[8];           12
   variable interrupt = interrupt_reg[4] : { ENABLE => '0', DISABLE => '1' };   13

   // Index register
   register index_reg = write base @ 2, mask '1..00000' : bit[8];               16
   private variable index = index_reg[6..5] : int(2);                           17

   register x_low  = read base @ 0, pre {index = 0}, mask '****....' : bit[8];  19
   register x_high = read base @ 0, pre {index = 1}, mask '****....' : bit[8];  20
   register y_low  = read base @ 0, pre {index = 2}, mask '****....' : bit[8];  21
   register y_high = read base @ 0, pre {index = 3}, mask '...*....' : bit[8];  22

   structure mouse_state = {                                                    24
      variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);       25
      variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);       26
      variable buttons = y_high[7..5], volatile : int(3);                       27
   };
}
```

FIG. 1 – *Spécification de la souris bus Logitech*

fonction `get_xx` pour lire la variable, et une fonction `set_xx` pour lui affecter une valeur.
Dans la suite de cette section, nous présentons les constructions de base de Devil ainsi que leur utilisation, en décrivant en détails l'exemple de la souris bus Logitech (fig. 1), ainsi que des fragments de spécification provenant d'autres circuits. Toutes les fonctionnalités de Devil ne sont pas décrites dans cet article. Un rapport technique est disponible [15], qui décrit le langage dans son ensemble.

Ports.

Les ports sont les points de communication entre un contrôleur de périphérique et le processeur central. Le contrôleur offre souvent plusieurs ports, dont les adresses sont dérivées d'une ou plusieurs adresses de base. Ainsi, le constructeur de port, dénoté par l'opérateur `@`, prend comme arguments une adresse de base et un déplacement (par exemple `base @ 1`, ligne 4 de la figure 1). Pour permettre la vérification, l'intervalle des déplacements valides doit être précisé dans la déclaration de l'adresse de base (par exemple `port @ {0..3}`, ligne 1).

Registres.

Les registres définissent la granularité des échanges entre le processeur et le contrôleur. Par conséquent, la taille de chaque registre (en nombre de bits) doit être explicitement spécifiée. Les registres sont définis par deux ports : un port de lecture et un port d'écriture. Cependant, la lecture et l'écriture partagent souvent le même port. Dans ce cas, un seul port est spécifié, de même que pour un registre en lecture seule ou en écriture seule.
Une déclaration de registre peut être complétée par un masque, afin de préciser les conditions d'utilisation de chaque bit du registre. Ce masque est une séquence de caractères : un point dénote un bit utilisé par une variable, une étoile dénote un bit inutilisé dont la valeur est quelconque,

et un chiffre (0 ou 1) dénote un bit inutilisé dont la valeur est fixe (connue en lecture, imposée en écriture). Un registre peut avoir un masque différent en lecture et en écriture. Lorsqu'aucun masque n'est spécifié, le registre est considéré comme ayant tous ses bits utilisés. Voici un exemple de masque, tiré de la figure 1 (ligne 16) :

```
register index_reg = write base @ 2, mask '1..00000' : bit[8];
```

Ce masque indique que seuls les bits 6 et 5 sont utilisés. De plus, le bit 7 sera toujours positionné à 1 lors des écritures, et les bits 0 à 4 seront positionnés à 0.

Pré-actions des registres.

Sur certaines architectures, le nombre de ports adressables par le processeur est limité. Aussi, lorsqu'un contrôleur possède beaucoup de registres, ses concepteurs limitent souvent le nombre de ports nécessaires en multiplexant certains ports. Cette solution est aussi fréquemment adoptée lorsqu'un circuit est une version améliorée d'un modèle plus ancien : de nouveaux registres apparaissent, mais le nombre de ports doit rester identique pour ne pas compromettre la compatibilité. Lorsqu'un port est multiplexé, plusieurs registres lui sont associés, mais ils s'excluent mutuellement (un seul est actif à un instant donné). Généralement, la sélection du registre actif s'opère en modifiant une variable de contrôle. Cette étape de sélection, qui précède l'accès aux registres d'un port multiplexé, s'appelle en Devil une *pré-action* (dénotée par le mot-clé `pre`). Dans l'exemple de la souris bus, les deux registres `x_low` et `x_high` utilisent le même port `base @ 0` (lignes 19 et 20 fig. 1). C'est la variable `index` qui, suivant qu'on lui affecte 0 ou 1, sélectionne l'un ou l'autre des deux registres.

```
register x_low  = read base @ 0, pre {index = 0}, mask '****....' : bit[8];
register x_high = read base @ 0, pre {index = 1}, mask '****....' : bit[8];
```

Variables de contrôle.

Afin de minimiser le nombre des échanges avec le processeur, les concepteurs de contrôleurs ont tendance à placer plusieurs données indépendantes dans un même registre. Le programmeur doit alors combiner ces données à chaque écriture du registre, et les séparer les unes des autres à chaque lecture. Il arrive aussi qu'une donnée de grande taille soit répartie sur plusieurs registres : il faut alors reconstituer cette donnée en concaténant les différents morceaux. Toutes ces opérations s'expriment en C à l'aide d'opérateurs bit à bit (décalages et combinaisons logiques), peu lisibles car de très bas niveau. En Devil, chaque donnée indépendante est une entité à part entière, appelée variable de contrôle, qui reçoit un identificateur et un type. La façon dont une variable est contruite s'exprime directement en termes de fragments de registres, le compilateur Devil se chargeant de générer le code adéquat. Quant au typage, il permet de contrôler que la variable est correctement utilisée (on ne peut lui affecter que des valeurs du même type, par exemple). Les types possibles sont les booléens, les ensembles et intervalles d'entiers, et surtout les types énumérés, qui sont parlants et améliorent beaucoup la lisibilité. Reprenons l'exemple du registre `index_reg` (voir le haut de la page) et examinons son masque. Tous les bits marqués d'un point (et seulement ceux-là) doivent servir à définir des variables. Dans le cas de `index_reg`, seuls les bits 6 et 5 sont utilisés. Ils constituent la variable `index` (ligne 17 fig. 1) ayant pour type "entier de 0 à 3" (entier non signé sur deux bits). Voici sa définition :

```
private variable index = index_reg[6..5] : int(2);
```

L'attribut `private` signifie que cette variable ne fera pas partie de l'interface fonctionnelle du contrôleur. En d'autres termes, son identificateur ne sera pas visible depuis les couches supérieures du pilote.

La variable `dx` (ligne 25 fig. 1) est un autre exemple intéressant, car elle est construite par concaténation de deux morceaux de registres.

```
variable dx = x_high[3..0] # x_low[3..0], ...
```

Types énumérés

Comme mentionné au paragraphe précédent, le typage des variables de contrôle permet de sécuriser leur utilisation. Les types énumérés sont particulièrement précieux : ils fournissent un identificateur pour chaque valeur possible de la variable, ce qui limite les risques d'erreur en améliorant la lisibilité. Les symboles `<=`, `=>` and `<=>` définissent des constantes en lecture, écriture, et lecture-écriture (respectivement). Par exemple, la variable `config` (ligne 9 fig. 1) est en écriture seule, et ses valeurs possibles sont `CONFIGURATION` and `DEFAULT_MODE`:

```
variable config = cr[0] : {
   CONFIGURATION => '1', DEFAULT_MODE  => '0' };
```

Attributs comportementaux.

Avoir plusieurs variables de contrôle dans le même registre pose des problèmes de cache et de synchronisation.
Soit un registre contenant deux variables $v1$ et $v2$. Chaque fois que le programmeur du pilote affecte une valeur à $v1$, la couche basse est obligée d'écrire le registre en entier. Cela suppose de trouver quelque chose à mettre dans $v2$. Heureusement, toutes les variables ont une valeur par défaut. Pour obtenir celle de $v2$, la couche basse doit d'abord savoir si l'affectation de $v2$ est idempotente. Autrement dit, elle doit savoir si affecter deux fois de suite la même valeur à $v2$ a exactement le même effet qu'une seule affectation. Quand rien n'est précisé dans sa définition, une variable est supposée avoir la propriété d'idempotence. Dans ce cas, la valeur par défaut est la dernière valeur affectée à la variable, mémorisée dans un cache par la couche basse. Le mot-clé `trigger`, ajouté à la définition d'une variable, indique que les affectations ne sont pas idempotentes. Dans ce cas, le principe du cache ne fonctionne pas. Cependant, les concepteurs d'un contrôleur prévoient souvent une valeur neutre pour chaque variable `trigger`, c'est-à-dire une valeur pour laquelle l'affectation de la variable est sans effet. Quand une valeur neutre existe, elle constitue une valeur par défaut idéale. Devil permet de la mentionner grâce au mot-clé `except`. Seules les variables `trigger` dotées d'une valeur neutre peuvent partager leur registre avec d'autres variables.
Le comportement des variables en lecture a aussi son importance. Par défaut, lire une variable est une opération idempotente, c'est-à-dire que deux lectures successives (sans affectation entre les deux) renvoient toujours la même valeur. Le mot-clé `volatile` permet d'indiquer les variables n'ayant pas cette propriété.
L'exemple qui suit, extrait de la spécification du contrôleur réseau 8390, illustre l'utilisation de `trigger` et `volatile`. La variable `transmitPacket` est une commande : lui affecter la valeur `TRANSMIT` déclenche la transmission d'un paquet. Quant à `transmittingPacket`, cette variable en lecture seule vaut `true` lorsqu'un paquet est en cours de transmission. Son contenu passe spontanément à `false` dès que la transmission est terminée.

```
register CommandReg = ...
variable transmitPacket = write CommandReg[2], trigger except DO_NOTHING :
{
  TRANSMIT   => '1',
  DO_NOTHING => '0'
};
variable transmittingPacket = read CommandReg[2], volatile : bool;
```

Structures.

Reprenons l'exemple de deux variables de contrôle $v1$ et $v2$. Si elles ne sont pas dans le même registre, l'une est forcément lue avant l'autre. Par contre, si elles sont dans un même registre, lire ce registre est une opération atomique, capable de récupèrer en même temps les contenus de $v1$ et $v2$. Cependant, cette simultanéité potentielle n'est pas exploitable par le programmeur du pilote, pour qui $v1$ et $v2$ sont des variables indépendantes. C'est rarement un problème, mais si $v1$ et $v2$ ont à la fois un comportement `volatile` et un lien sémantique entre elles, la cohérence du

couple (v1, v2) peut nécessiter deux valeurs provenant du même instant atomique. Pour de tels cas particuliers, Devil permet d'exprimer la simultanéité en regroupant plusieurs variables dans une même entité appelée `structure`. Un exemple est donné par les variables `dx`, `dy` et `buttons` (lignes 24 à 27 fig. 1).

```
structure mouse_state = {
   variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
   variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
   variable buttons = y_high[7..5], volatile : int(3);
};
```

Pour cet exemple, le compilateur Devil crée quatre fonctions de lecture : `get_mouse_state`, `get_dx`, `get_dy` et `get_buttons`. La première concerne la structure tout entière. Son rôle est de lire les registres, et de placer le contenu des variables dans un cache. Les trois dernières fonctions concernent les champs de la structure. Elles n'accèdent pas au contrôleur, et se contentent de renvoyer les valeurs stockées dans un cache par `get_mouse_state`. Ainsi, bien que le registre `y_high` contienne deux variables, il n'est lu qu'une seule fois.

Sérialisation des registres

Le contrôleur DMA 8237A fournit des compteurs 16 bits, mais un seul port 8 bits est alloué à chacun d'eux. Comme le montre l'exemple qui suit, construire le compteur `x` nécessite la concaténation des deux registres `cnt_high` et `cnt_low`. Ces registres utilisent le même port, mais ne peuvent pas être sélectionnés indépendamment l'un de l'autre. Le mécanisme est plus subtil : écrire dans la variable `flip_flop` sélectionne `cnt_low`, et `cnt_high` est disponible dès que `cnt_low` est lu. Pour modéliser ce mécanisme, il faut pouvoir spécifier l'ordre dans lequel les registres sont lus avant d'être concaténés. Le mot-clé `serialized as` offre cette possibilité. Il est suivi d'une liste de registres, triée dans l'ordre chronologique des accès.

```
register cnt_low = data, pre {flip_flop = *} : bit[8];
register cnt_high = data : bit[8];
variable x = cnt_high # cnt_low : int(16) serialized as {cnt_low; cnt_high};
```

Accès multiples.

Certains contrôleurs comportent un tampon interne, avec une seule variable pour accéder à ce tampon. Dans ce cas, chaque lecture de la variable retire un élément au tampon, permettant ainsi de le vider élément par élément. De façon similaire, des écritures successives dans la variable ajoutent des éléments au tampon et permettent de le remplir.
Pour le programmeur du pilote, vider ou remplir le tampon se traduit par une boucle en C contenant un simple accès à une variable. Or, le compilateur C ne sait pas toujours optimiser parfaitement une telle boucle. Par exemple, sur une architecture de type x86, l'idéal est d'utiliser l'instruction assembleur `rep`. Cette instruction exécute la boucle au niveau du microcode, évitant ainsi les tests et les branchements. Pour exploiter au mieux l'architecture cible, le compilateur Devil sait générer des fonctions optimisées qui effectuent plusieurs accès par appel. Elles évitent au programmeur du pilote d'avoir recours à une boucle en C pour effectuer des accès multiples. Le mot-clé `block` sert à identifier les variables pour lesquelles ces fonctions particulières doivent être générées, en supplément des fonctions d'accès habituelles. L'exemple de la variable `Ide_data`, tiré de la spécification du contrôleur IDE, illustre l'utilisation de `block`.

```
variable Ide_data =
   ide_data, trigger, volatile, block : int(16);
```

3. Vérification de propriétés

Grâce à ses abstractions dédiées, Devil permet de décrire avec précision la sémantique de l'interface fonctionnelle d'un contrôleur de périphérique. Parce qu'elle est explicite, cette sémantique permet

des contrôles de cohérence qui sont hors de portée d'un langage généraliste tel que C. Cette vérification en profondeur permet de détecter les erreurs beaucoup plus tôt dans le processus de développement. La robustesse des pilotes s'en trouve améliorée. Nous avons d'ailleurs pu le vérifier en comparant les processus de développement par une analyse de mutations, décrite dans un autre article [13].
Cette section résume les propriétés vérifiables, soit par le compilateur Devil (cohérence de la spécification), soit par le compilateur C (utilisation correcte de la couche basse générée).

3.1. Vérification des spécifications

En raison de la nature déclarative du langage Devil, de multiples propriétés peuvent être vérifiées pour s'assurer de la cohérence d'une spécification. En voici quelques exemples.

Typage.

En Devil, les entités (ports, registres, et variables) sont fortement typées : chaque utilisation d'une entité doit être conforme au type déclaré dans la définition de cette entité. Les types peuvent exprimer des contraintes sur l'accès à une entité (lecture seule, écriture seule), sur sa taille en bits, ou sur l'ensemble des valeurs possibles.

Absence d'omission.

Tous les ports et les registres déclarés dans une spécification Devil doivent être utilisées au moins une fois. De même, tous les bits marqués d'un point dans le masque d'un registre (cf. section 2) doivent être utilisés pour contruire une variable.

Absence de redéfinition.

Dans une spécification Devil, chaque entité ne peut être déclarée qu'une seule fois. Cette contrainte concerne les ports, les registres, les variables, les types énumérés, et les éléments de type (symboles et motifs de bits).

Absence de définition conflictuelle.

Les définitions des registres et des variables doivent être disjointes. Plus précisément, deux registres ne peuvent pas être définis sur le même port, sauf s'ils ont des pré-actions différentes, des masques disjoints, ou des restrictions d'accès opposées (l'un en lecture seule, l'autre en écriture seule). De même, aucun bit d'un registre ne peut être utilisé pour définir deux variables différentes.

3.2. Vérification des pilotes

Dans le code du pilote, il est possible de vérifier que la couche basse générée par le compilateur Devil est correctement utilisée. Cette vérification peut être à la fois statique (effectuée par le compilateur C) ou dynamique (des tests sont insérés dans le code et activés à l'exécution). Afin de ne pas compromettre l'efficacité du code, ces deux types de vérification sont optionnels.
Lors de l'affectation d'une variable de contrôle, il est possible de vérifier que la valeur affectée fait bien partie des valeurs permises par le type de la variable. Si la valeur est constante, ce test peut généralement être effectué par le compilateur C, lors de la compilation du pilote. Cependant, le système de typage de C n'est pas assez puissant pour vérifier correctement tous les types de Devil. Pour certains d'entre eux, un test pendant l'exécution est nécessaire. Tester pendant l'exécution permet aussi de vérifier le type du contenu d'une variable que l'on vient de lire (au cas où le contrôleur n'aurait pas un comportement conforme à sa spécification).
Notre expérience de ré-écriture des pilotes pour Devil nous a montré que les tests dynamiques permettent de détecter très tôt les erreurs d'utilisation, avant qu'elles ne deviennent des bogues insidieux [13]. C'est particulièrement intéressant pour les pilotes en mode noyau, car ils sont difficiles à déboguer interactivement. De plus, comme les tests dynamiques sont automatiquement et systématiquement insérés ou enlevés par le compilateur, leur emploi est à la fois simple et sûr.

4. Performances

Toutes les entrées-sorties d'un système passent par les pilotes de périphériques. Par conséquent, une part significative des performances globales d'un système dépend des performances des pilotes. Bien que Devil facilite le développement de pilotes robustes, il serait inutilisable s'il ne permettait pas d'écrire des pilotes performants.
Pour savoir si Devil occasionne un surcoût à l'exécution, nous avons ré-implémenté à l'aide de Devil plusieurs pilotes existants. Ensuite, nous avons comparé les performances entre les pilotes modifiés et les pilotes originaux. Pour que l'expérience soit significative, nous avons choisi trois périphériques différents, réputés être avides de performances : un contrôleur disque ATA/IDE, une carte graphique, et une carte réseau NE2000.
Dans la suite de cette section, nous discutons des problèmes de performances que peut poser l'emploi de Devi, puis nous présentons les résultats pour chacun des trois pilotes. Il est à noter que nos tests de performances sur le pilote IDE et le pilote graphique sont décrits avec plus de détails dans l'un de nos récents articles [9]. Nos expérimentations sur le pilote NE2000 sont plus récentes, et elles confirment les résultats obtenus sur les deux autres pilotes.

Surcoûts potentiels.

Lorsque les tests dynamiques sont désactivés, le compilateur Devil génère un code optimisé, et conçu pour être facilement optimisable par le compilateur C. De plus, les fonctions d'accès générées sont implémentées sous la forme de fonctions `inline`, afin que leur appel n'occasionne aucun surcoût.
Dans la section 2, nous avons vu que le compilateur Devil pouvait générer des fonctions d'accès multiples. Ne pas utiliser ces fonctions à la place de boucles en C peut dégrader les performances.
Devil permet de regrouper plusieurs variables d'un même registre dans une structure, lorsque la cohérence des données exige un accès simultané aux variables (voir section 2). En principe, en l'absence d'une telle contrainte, les variables d'un même registre restent indépendantes, afin que leur utilisation soit souple et élégante. Cependant, accéder à plusieurs de ces variables provoque alors plusieurs accès au registre, ce qui est une perte potentielle de performances. En pratique, ce problème est surtout gênant pour un registre dont les variables sont systématiquement utilisées ensemble. Cependant, ce cas révèle souvent un choix délibéré de la part du concepteur du contrôleur : les variables sont dans le même registre pour en faciliter l'accès, et pas seulement pour gagner de la place. Elles ont généralement un rapport sémantique entre elles, et les grouper dans une structure ne compromet donc pas la lisibilité du code.

Pilote IDE.

Le pilote IDE provient de Linux 2.2-12. Nous l'avons testé en mode DMA et dans plusieurs mode PIO, en faisant varier la largeur des accès (16 ou 32 bits) et le nombre de secteurs. Le débit a été mesuré avec `hdparm`, une commande standard permettant de mesurer les performances du système. Nous n'avons constaté aucune différence de débit entre le pilote original et celui utilisant Devil. On peut tout de même noter que les fonctions d'accès multiples générées par le compilateur Devil s'avèrent ici indispensables : les remplacer par des boucles en C enlève 10% de débit au mode PIO. Le tableau ci-dessous donne le détail des résultats.

Mode de transfert	DMA	PIO					
Secteurs par interrupt.	-	16		8		1	
Largeur des accès	-	32 bits	16 bits	32 bits	16 bits	32 bits	16 bits
Pilote original							
Débit en Mo/s	14.25	8.17	4.45	8.09	4.42	6.93	4.06
Pilote Devil sans boucles en C							
Débit en Mo/s	14.25	8.17	4.45	8.09	4.42	6.93	4.06
Pilote Devil avec boucles en C							
Débit en Mo/s	14.25	7.36	3.94	7.28	3.91	6.36	3.63

Pilote graphique X11.

Le pilote graphique provient de XFree86 3.3.6. Il est dédié au contrôleur graphique Permedia2 de 3DLabs. Pour minimiser le code dépendant du matériel, beaucoup de primitives sont dessinées

par le processeur. En fait, le pilote n'utilise le matériel que pour les deux primitives 2D les plus coûteuses : le rectangle plein et la copie d'une zone de l'écran.
Le débit a été mesuré grâce à l'utilitaire `xbench`. Le temps d'exécution d'une primitive par le contrôleur est proportionnel au nombre de pixels à afficher. Ainsi, le surcoût devient perceptible pour les très petites primitives (4 pixels). Le pire des cas que nous ayons mesuré est une dégradation de 6%, ce qui reste très raisonnable. Le tableau ci-dessous donne le détail des résultats en mode 16 bits/pixel. Avec 8 bits/pixel, les résultats sont similaires. Avec 24 et 32 bits/pixel, ils sont meilleurs (3% de dégradation dans le pire des cas).

Taille de la primitive (pixels)	2 x 2	10 x 10	100 x 100	400 x 400
Test du rectangle				
Débit pilote original (rectangles/s)	982338	333670	21022	2221
Débit pilote Devil (rectangles/s)	945916	332499	21033	2221
Rapport débits Devil/original	96%	100%	100%	100%
Test de la copie de bloc				
Débit pilote original (copies/s)	145084	85994	3502	238
Débit pilote Devil (copies/s)	136755	85561	3512	238
Rapport débits Devil/original	94%	99%	100%	100%

Pilote Ethernet

Comme le pilote IDE, le pilote Ethernet provient de Linux. Il est dédié aux cartes compatibles NE2000. En mesurant le débit avec `ttcp`, nous n'avons pas constaté de différence significative entre le pilote original et celui utilisant Devil.

	Transmission	Réception
Débit du pilote original (ko/s)	1108	1010
Débit du pilote Devil (ko/s)	1112	1006

5. Bibliothèque de spécifications

Tout au long de cet article, et en particulier à la section 3, nous avons insisté sur les bénéfices apportés par l'utilisation de Devil dans le processus de développement d'un pilote. Nous sommes convaincus que, même pour un programmeur partant uniquement de la documentation d'un circuit, écrire une spécification Devil avant de s'attaquer aux couches supérieures du pilote rend le développement plus facile, plus court, et plus sûr. En effet, la phase d'écriture de la spécification permet au programmeur de se concentrer une bonne fois pour toutes sur les subtilités de l'interface du circuit, sans être confronté dès le début à des problèmes de bogues insidieux ou de choix d'implémentation. Une fois les fondations générées par le compilateur Devil, le programmeur peut bâtir dessus les couches supérieures du pilote, en se concentrant uniquement sur la communication avec le système, et sur l'interface fonctionnelle du circuit (et non plus les mécanismes sous-jacents).

Cependant, nous sommes conscients que le gain en productivité est bien plus spectaculaire lorsque la spécification Devil est déjà écrite, et que le programmeur du pilote n'a plus qu'à la compiler pour son architecture. Parce qu'elle définit clairement le type et le comportement des variables d'un contrôleur de périphérique, une spécification Devil joue le rôle d'une base d'information sur l'utilisation correcte de ce contrôleur. En fait, les noms des variables de contrôle constituent une interface fonctionnelle qui guide le programmeur du pilote. Le développement est simplifié, et la lisibilité du code est améliorée. De plus, une vérification automatique est possible, à deux étapes du développement : sur la cohérence de la spécification Devil d'une part, sur l'utilisation correcte de la bibliothèque générée d'autre part.
Or, un contrôleur de périphérique est souvent utilisé dans des contextes très variés : différentes architectures, et surtout de nombreuses applications embarquées. À chaque nouveau contexte, il faut écrire un nouveau pilote. Pourtant, l'interface du circuit reste la même. Il y aurait donc beaucoup à gagner si chaque spécification Devil, une fois écrite, était aussitôt mise à la disposition de tous.
Dans l'idéal, les constructeurs de circuits pourraient même compléter leur traditionnelle documentation par une spécification Devil, offrant ainsi à leurs clients un ensemble prêt à l'emploi,

constitué d'un composant matériel et d'un composant logiciel.

Pour commencer à concrétiser cela, nous avons mis en place une bibliothèque de spécifications (encore très modeste), accessible à l'adresse suivante :

http://compose.labri.fr/prototypes/devil/specs/

On peut y trouver toutes les spécifications mentionnées dans cet article, et nous invitons vivement les bonnes volontés à y contribuer.

6. Travaux connexes

Nos recherches sur les pilotes ont débuté avec une étude sur les cartes graphiques et leur support dans X11. Nous avons alors conçu un langage, nommé GAL, qui permettait de générer un pilote X11 complet à partir de la spécification d'une carte graphique [16]. Bien que montrant parfaitement la pertinence du concept, GAL avait un domaine d'application très restreint.

Le projet UDI (*Uniform Driver Interface*), lancé conjointement par les principaux fournisseurs de plateformes Unix, a pour but de rendre les pilotes (ou plus exactement leurs sources) portables d'un Unix à l'autre. A cette fin, UDI normalise l'API entre le noyau du système et la couche supérieure des pilotes [11]. Contrairement à Devil, UDI concerne la couche supérieure des pilotes, et non les interactions de bas niveau avec le matériel. Néanmoins, un tel projet prouve que l'amélioration du processus de développement des pilotes est un problème d'actualité.

WinDK de BlueWater System [3] et DriverWorks de NuMega [4] sont des générateurs de pilotes spécifiques à Windows. Ces deux outils sont capables de générer un squelette de pilote, dont les caractéristiques sont spécifiées à l'aide d'une interface graphique. Afin de compléter facilement ce squelette, le programmeur du pilote dispose aussi d'une bibliothèque de classes C++ de granularité élevée, qui cache les appels système sous une API de plus haut niveau. Cependant, à l'instar du projet UDI, ces générateurs laissent de côté le problème des contrôleurs de périphériques et de leur interface.

Les langages de spécification de circuits intégrés existent depuis des années. Le standard VHDL [8], très utilisé dans ce domaine, est l'un des plus expressif. Il couvre plusieurs aspects de la conception des circuits, tels que la documentation, la simulation, et la synthèse. Le langage VHDL offre à la fois des abstractions de haut niveau et de bas niveau. Par exemple, le programmeur dispose de tableaux et de boucles, ainsi que de littéraux de type vecteur de bits et d'un opérateur d'extraction de bits. Cependant, toutes les abstractions de VHDL ont pour but de spécifier les rouages internes du circuit, par son interface de programmation. Par conséquent, cette interface n'est pas dénotée explicitement dans la spécification VHDL, ce qui empêche le compilateur d'en vérifier la cohérence. Une fonctionnalité intéressante de VHDL est de pouvoir attacher une chaîne de caractères à chaque variable. Il serait possible d'utiliser cet attribut pour incorporer dans la spécification d'un circuit des informations spécifiques à son interface de programmation. Toutefois, pour être vraiment utiles, ces informations textuelles devraient avoir une syntaxe normalisée et être prises en compte par le compilateur. En quelque sorte, cela reviendrait à inclure les concepts de Devil dans VHDL.

Enfin, la *N*ew Jersey Machine-Code Toolkit [12] aide les programmeurs à écrire des applications qui manipulent du code machine. A partir de la spécification d'un jeu d'instructions, cet outil génère une bibliothèque de niveau assembleur, dont les fonctions lisent ou écrivent du code binaire. L'outil effectue aussi des vérifications simples au niveau de la spécification.

7. Conclusion

Cet article a présenté Devil, un langage servant à spécifier l'interface de programmation des contrôleurs de périphériques. Son expressivité est démontrée par la variété des spécifications que nous avons déjà écrites : contrôleurs de souris, de DMA, d'interruptions, de carte graphique, de carte Ethernet, et de disque IDE. Dans cet article, nous avons aussi proposé des outils permettant de vérifier la cohérence des spécifications Devil, de les exploiter pour générer automatiquement la couche basse des pilotes, et de vérifier l'utilisation correcte de cette couche. Ces outils améliorent

ainsi la *robustesse* des pilotes, une qualité recherchée depuis longtemps. Nous avons aussi montré, à l'aide de résultats expérimentaux, que l'utilisation de Devil ne grevait pas les performances des pilotes. Enfin, nous avons proposé la mise en place d'une bibliothèque en ligne de spécifications, et encouragé la communauté à y contribuer.

Remerciements.
Ces travaux ont été partiellement supportés par France Telecom (contrat CTI 991B726), le Ministère Français de la Recherche (contrat Phenix 99S0362), et le Ministère Français de l'Éducation Nationale.

Bibliographie

1. Bershad (B.N.), Anderson (T.E.), Lazowska (E.D.) et Levy (H.M.). – Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, vol. 8, n° 1, février 1990, pp. 37–55.
2. Birrell (A.) et Nelson (B.). – Implementing remote procedure calls. *ACM Transactions on Computer Systems*, vol. 2, n° 1, février 1984, pp. 39–59.
3. BlueWater Systems, Inc. – *WinDK Users Manual*. http://www.bluewatersystems.com/.
4. Compuware NuMega. – *DriverWorks User's Guide*. http://www.numega.com/.
5. Dekker (E. N.) et Newcomer (J. M.). – *Developing Windows NT device drivers : A programmer's handbook*. – Addison-Wesley, mars 1999, première édition.
6. Draves (R.), Jones (M.) et Thompson (M.). – *MIG - The MACH Interface Generator*. – School of Computer Science, Carnegie Mellon University, juillet 1989.
7. Eide (E.), Frei (K.), Ford (B.), Lepreau (J.) et Lindstrom (G.). – Flick: A flexible, optimizing IDL compiler. *In : Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 44–56. – Las Vegas, NV, USA, juin 15–18, 1997.
8. IEEE Standards. – *1076-1993 Standard VHDL Language Reference Manual*, 1994. http://standards.ieee.org/.
9. Mérillon (F.), Réveillère (L.), Consel (C.), Marlet (R.) et Muller (G.). – Devil: An IDL for Hardware Programming. *In : 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*. pp. 17–30. – USENIX Association.
10. O'Malley (S.), Proebsting (T.) et Montz (A.B.). – USC: A universal stub compiler. *In : Proceedings of Conference on Communication Architectures, Protocols and Applications*. – London (UK), septembre 1994.
11. Project UDI. – *UDI Specifications, Version 1.0*, September 1999. http://www.project-udi.org/.
12. Ramsey (Norman) et Fernandez (Mary F.). – The New Jersey machine-code toolkit. *In : Proceedings of the Winter USENIX Conference*. – New Orleans, LA, janvier 1995.
13. Réveillère (L.) et Muller (G.). – *Improving Driver Robustness: and Evaluation of the Devil Approach*. – Rapport de recherche n° 1385, IRISA, mars 2001. À paraître dans les actes de la conférence DSN 2001.
14. Rubini (A.). – *Linux Device Drivers*. – O'Reilly, février 1998, première édition.
15. Réveillère (L.), Mérillon (F.), Consel (C.), Marlet (R.) et Muller (G.). – *The Devil Language*. – Research Report n° 1319, Rennes, France, IRISA, mai 2000.
16. Thibault (S.), Marlet (R.) et Consel (C.). – Domain-Specific Languages: from Design to Implementation – Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, vol. 25, n° 3, mai–juin 1999, pp. 363–377.