Sauvegarde des variables locales en CPC

Matthieu Boutier

Juin 2011

Table des matières

Introduction

Deux choses peuvent désigner « CPC » : le langage CPC, une extension du C conçu pour la concurrence de *threads* coopératifs, et le compilateur associé, qui transforme un code CPC en style à *thread* en un code C équivalent en style à événements. La définition du langage, l'écriture du compilateur et la preuve de la correction des méthodes employées ont été effectuées par Juliusz Chroboczek et Gabriel Kerneis.

Le compilateur existant transforme le code CPC en un code C équivalent écrit en style à événement. Ceci s'obtient en effectuant une série de passes successives, prouvables et prouvées, dont l'une d'elles est la transformation en style par passage de continuation. Cette transformation est rendue possible par une technique appelée *lambda-lifting*, peu courante voire inconnue pour les programmes impératifs. Si les résultats effectués sur CPC étaient probants, la conjecture de son efficacité particulière due à ces méthodes demandait à être vérifiée : une alternative possible était l'utilisation d'environnements, et il n'était pas impossible que ceux-ci soient vraiment plus rapides.

Le but de cette étude est donc d'une part la réécriture d'une partie du compilateur CPC en introduisant des environnements, et d'autre part la quantification du coût qu'offre une telle implémentation par rapport à CPC. Le nouveau compilateur s'appelle E-CPC.

Préliminaire

Concurrence. La concurrence est omniprésente dans la plupart des programmes : par exemple, le client de messagerie électronique doit permettre à l'utilisateur d'écrire une lettre, donc de recevoir des informations du clavier, et d'afficher les caractères à l'écran, de récupérer le courrier, le lisant depuis l'Internet et l'écrivant sur le disque, etc. Plusieurs *flots de contrôle* s'exécutent en parallèle : on dit alors qu'ils sont *concurrents*, par opposition aux programmes séquentiels.

Dans un programme séquentiel, il y a un unique flot de contrôle, mais dans un programme concurrent, il en faut plusieurs : pour cela le plus courant est d'utiliser les abstractions de flots de contrôle que sont les processus et les *threads*. La différence entre ces deux abstractions est qu'un processus dispose de sa propre mémoire, alors qu'un *thread* partage sa mémoire avec les autres *threads*. Les deux abstractions disposent de leur propre *pile* d'appels [ASU88], à savoir une zone mémoire servant à enregistrer l'état courant d'un processus ou d'un *thread* lors d'un appel de fonction : lorsque la fonction retournera, le processus ou le *thread* en question retrouvera les informations propres à la fonction appelante. On appelle cette zone mémoire *pile* car plusieurs appels successifs de fonctions sont possibles : à chaque appel de fonction, les données sont enregistrées au *sommet* de la pile, et à chaque terminaison de fonction, les données sont lues et retirées du sommet de la pile.

Pour pouvoir exécuter plusieurs flots de contrôle vraiment en parallèle, il serait nécessaire d'avoir un processeur par flot de contrôle à exécuter ; cependant, on peut simuler la concurrence en utilisant un gestionnaire, appelé *ordonnanceur*, qui distribue à chaque flot de contrôle l'accès au processeur. L'ordonnanceur peut gérer la concurrence de deux manières : préemptive, ou coopérative. Un ordonnanceur *préemptif* allouera à chaque flot de contrôle un certain quantum de temps, pas nécessairement le même, au bout duquel il sera interrompu. Un ordonnanceur *coopératif* donnera l'accès au processeur à un unique flot de contrôle, tous les autres étant en attente, jusqu'à ce que celui-ci accepte de rendre la main. On dit alors qu'il *coopère*, et l'endroit où il le fait est appelé *point de coopération*. Un flot de contrôle qui coopère ne termine pas nécessairement : il continuera alors son exécution lorsque l'ordonnanceur lui redonnera la main.

Exécuter des processus en parallèle peut donner lieu à un certain nombre de difficultés, en particulier en ce qui concerne les accès à la mémoire. Prenons par exemple l'extrait de pseudo-code suivant, qui lit la valeur d'un emplacement mémoire x et l'incrémente :

```
1. tmp = lecture (x);
2. écriture (x, tmp + 1);
```

Considérons deux processus P1 et P2 exécutant ce code en parallèle et tels que x leur soit commune et *tmp* propre. Si la valeur de x avant l'exécution de P1 et P2 était 0, on s'attendrait à ce que x vaille 2, or, comme dans le tableau ci-dessous, on se rend compte que si P1 exécute 1, puis que P2 exécute 1, alors chaque variable *tmp* propre aux deux processus vaudra 0, et chaque processus écrira dans x la valeur

0 + 1 (= 1), qui sera la valeur à la fin des deux exécutions de P1 et P2. Ainsi, le résultat obtenu est différent du résultat attendu. Le programmeur dispose de primitives de synchronisation, telles que les verrous, afin d'éviter de telles erreurs. Il faut néanmoins être précautionneux, car d'autres problèmes peuvent survenir, comme *l'interblocage* (deux processus attendant simultanément la ressource de l'autre processus).

étape	exécution	tmp_{P1}	tmp_{P2}	x
init.	_	_	_	0
1	P1.1	0	_	0
2	P2.1	0	0	0
3	P1.2	0	0	1
4	P2.2	0	0	1

Ces problèmes sont moindre pour un ordonnanceur coopératif : il n'exécute qu'un seul flot de contrôle à un moment donné. Le programmeur peut donc être assuré que les instructions séparant deux points de coopération seront exécutées de manière *atomique*, c'est-à-dire sans être interrompues. Cependant, l'ordonnanceur coopératif possède un risque de blocage dans le cas où un processus en cours d'exécution ne rend jamais la main.

Un ordonnanceur préemptif a l'avantage de s'assurer qu'aucun processus ne pourra bloquer les autres, et que, si le matériel le permet, plusieurs de ses processus pourront être exécutés en même temps, mais l'accès aux ressources est plus complexe.

Stratégies de gestion de la concurrence au sein d'un processus.

Plusieurs méthodes sont couramment utilisées pour faire de la concurrence : les *threads* et les événements. Les deux méthodes ne sont cependant pas nécessairement antagonistes [AHT+02]. Le langage CPC est en style à *threads*, mais le code C produit par le compilateur CPC est en style à événement, de plus, il permet l'utilisation parallèle de *threads* natifs. Je vais présenter ici les deux notions, et en extraire les intérêts et inconvénients.

Les *threads*. Dans un programme avec *threads*, le programmeur dispose de primitives permettant de synchroniser les *threads* entre eux. Chaque *thread* correspond à un flot de contrôle à exécuter en parallèle avec les autres. Le programmeur peut facilement se représenter le flot de contrôle correspondant à son code, car chaque *thread* est programmé séparément des autres, et les primitives de synchronisation ne coupent pas le code, mais font partie intégrante de celui-ci. De plus, si l'architecture le permet, plusieurs *threads* peuvent être exécutés parallèlement. En revanche, le passage d'un *thread* à l'autre peut être coûteux, de même que les synchronisations.

Les événements. La programmation avec événements se présente sous la forme d'une boucle principale, appelée *boucle à événements*, qui détecte les *événements* externes (entrées/sorties...) ou internes (alarmes...), et exécute des *gestionnaires d'événements*, associés à chaque événement, que le programme enregistre auprès de la boucle. Le traitement de chaque événement peut déclencher d'autres événements, permettant au calcul de se poursuivre. On ne veut pas, lorsqu'on fait de la concurrence, qu'un flot de contrôle monopolise l'accès au processeur pendant trop longtemps, sans quoi le système n'est plus assez réactif. Aussi, afin de rester responsif, il est nécessaire que la boucle à événement soit souvent exécutée, et donc que chaque fonction correspondant à l'exécution d'un événement soit relativement courte : elle s'exécute en effet de manière atomique par rapport aux autres gestionnaires.

Considérons par exemple un flot de contrôle qui attend un message, et, lorsqu'il est reçu, effectue une lecture sur disque, avant de répondre au message reçu. Dans un style à événement, il faudrait séparer ces deux activités, et créer des événements et gestionnaires d'événements correspondants. Tout d'abord, un événement pour dire qu'un message est reçu et le gestionnaire pour recevoir un message, avec la fonction à exécuter en cas de réception. Lorsque la boucle à événement détecte qu'un message est reçu, le gestionnaire d'événement est retiré de la boucle à événement, et la fonction exécutée. Celle-ci doit effectuer une lecture sur disque, mais ne doit pas bloquer, c'est pourquoi on introduit un événement pour dire qu'une lecture disque est terminée, et la fonction précédente ajoute un gestionnaire d'événement correspondant à la lecture disque au gestionnaire d'événement, avec une autre fonction associée.

L'inconvénient d'un tel style de programmation est que le flot de contrôle est découpé en plusieurs gestionnaires d'événements, rendant le code difficile à lire et à corriger. En revanche, il n'y a pas de changement de contexte, comme pour les *threads*, ce qui est source d'efficacité.

Techniques de compilation employées par CPC

Le compilateur CPC transforme un programme CPC, écrit en style à *thread*, en programme C à style par événement. Pour cela, il utilise les notions de continuation, et de conversion CPS.

Les continuations. La *continuation* d'un fragment de code est intuitivement ce qu'il reste à faire après ce code. On peut utiliser les continuations pour mettre en œuvre une stratégie de gestion de la concurrence par événements : il suffit que la boucle à événement dispose des continuations des exécutions des différents processus, et qu'elle exécute à tour de rôle un morceau de telle ou telle continuation (on parle alors de *continuation courante*). L'implémentation de la concurrence en utilisant des continuations est courante dans les langages fonctionnels, mais elle ne l'est pas à ma connaissance dans les langages impératifs.

Conversion CPS. On peut convertir un code en style à *thread* en un code à style par passage de continuation : ceci s'appelle la conversion CPS.

On peut comprendre la conversion CPS en la considérant comme une machine à pile, modélisation proche de ce qui se passe dans CPC. Prenons l'exemple suivant, où on évalue `double( succ(7) )` avec `double` la fonction qui double son argument, `succ` la fonction qui incrémente son argument, et ϵ la pile vide :

Opération	Pile
`double( succ(7) )`	ϵ
`succ(7)`	`double` . ϵ
`8`	`double` . ϵ
`double( 8 )`	ϵ
`16`	ϵ

En effet, la fonction `double` a besoin d'un nombre comme argument, ce que n'est pas `succ(7)` (c'est une fonction appliquée à un nombre), on se souvient donc qu'il faudra appliquer la fonction double au résultat de `succ(7)` en *poussant* `double` sur la pile. En fait, la pile est la continuation du calcul courant, ainsi en ligne 2 `double` est la continuation de `succ(7)`.

On appelle *fonction CPS* (« *continuation passing style* ») une fonction écrite en style par continuation : elle prend en argument sa propre continuation, et peut la modifier pour stocker la suite du calcul, avant de rendre la main à l'appelant (concrètement, la boucle à événement). Dans notre exemple, la conversion CPS d'une fonction exécutant `double( succ (7) )` serait une fonction qui pousserait sur la continuation la fonction `double`, puis exécuterait `succ(7)`.

1 CPC

1.1 Le langage CPC

Description brève. Le langage CPC est le langage C étendu par les mots-clés « `cps` », « `cpc_spawn` » et « `cpc_attached` », ainsi que quelques primitives pour la concurrence, et de la possibilité de définir des fonctions internes.

Par abus de langage et sans indication explicite, nous appellerons aussi fonction CPS les fonctions CPC précédées du mot-clé `cps`, car elles sont destinées à devenir des fonctions CPS à proprement parler.

Dans CPC, les fonctions CPS (« *continuation passing style* ») sont les seules qui sont interruptibles, et elles ne peuvent être appelées que par une fonction CPS (ou par l'ordonnanceur — la boucle à événement).

On appelle fonction *CPS-convertible* une fonction qui est constituée d'une première partie de code non-préemptible, suivie d'appels à des fonctions CPS.

Nouveaux mots-clés. Le mot-clé « `cps` » sert donc à déclarer quelles sont les fonctions qui seront transformées par la conversion CPS ; il précède l'entête et le prototype de la fonction. Par exemple :

```
cps int wonder(int x, int y);
[...]
cps int wonder(int x, int y) {
  return x + y;
}
```

Le mot-clé « `cpc_spawn` » permet de lancer le code d'une fonction CPS dans un nouveau *thread*, par exemple :

```
{...
  cpc_spawn wonder(x,y);
  ...
}
```

Il peut être intéressant de vouloir exécuter une partie du programme dans un *thread* natif (par exemple pour l'utilisation des multi-cœurs, ou à cause de longs calculs qui pourraient ralentir la réactivité de l'ordonnanceur, ou encore afin d'utiliser des APIs bloquantes), aussi il est possible d'assigner la continuation courante à un *thread pool* (unité de regroupement de *threads*) avec la fonction `cpc_attach`, qui prend en paramètre le nouveau *thread pool*, et qui renvoie l'ancien. Pour réaffecter une continuation à l'ordonnanceur de CPC, il suffit de lui donner `cpc_default_scheduler` comme argument.

Voici un exemple, tiré de Hekate [AC09], dans lequel on désire faire un `getaddrinfo`. Cette fonction est bloquante, on choisit donc demander à la continuation courante d'exécuter cette fonction dans un *thread* natif lié au *thread pool* par défaut de CPC, puis de revenir à son état précédent (en particulier, si `s = cpc_default_scheduler`, alors la continuation reviendra dans l'ordonnanceur de CPC) :

```
cpc_scheduler *s = cpc_attach(cpc_default_threadpool);
rc = getaddrinfo(name, ...);
cpc_attach(s);
return rc;
```

Le mot-clé « cpc_attached » sert à simplifier cette procédure en prenant en paramètre une expression et un bloc d'instructions, à savoir respectivement le *thread pool* qu'on veut associer, et la fonction à associer. Le code ci-dessus deviendrait donc par exemple :

```
cpc_attached(cpc_default_threadpool) {
  rc = getaddrinfo(name, ...);
}
return rc;
```

Comme cpc_attached est capable de gérer les sorties non-locales, on peut simplifier le code ci-dessus en :

```
cpc_attached(cpc_default_threadpool) {
  return getaddrinfo(name, ...);
}
```

Des macros et quelques autres primitives existent afin de faciliter la gestion de ce genre d'opérations [CK10].

Fonctions internes. Une autre différence avec le langage C est la possibilité de définir des fonctions internes, avec la restriction qu'il ne s'agit que de fonctions CPS, et que les variables libres de ces fonctions sont des copies des variables des fonctions englobantes (sauf lorsque celles-ci sont static). Ces fonctions sont en pratique souvent implicites, car cachées par le sucre syntaxique offert par CPC, comme par exemple dans le code suivant, qui transforme le bloc d'instructions passé en paramètre à cpc_spawn en fonction interne :

Sucre syntaxique	Code équivalent

`void f()`
`{`
`  int y = 5;`
`  cpc_spawn {`
 	
`  int x;`
`  x = 4;`
`  printf("%d", x + y);`
`  };`
`}`
	`void f()`
	`{`
	`  int y = 5;`
	`  cpc_spawn aux();`
	`  cps void aux (void) {`
	`  int x;`
	`  x = 4;`
	`  printf("%d", x + y);`
	`  }`
	`}`

Il est important de comprendre que les variables libres des fonctions internes sont passées par copie à la fonction interne, au moment de chaque appel. Voici quelques essais de modification de la valeur d'une variable de la fonction englobante par une fonction interne, les deux premiers ne donnant pas le résultat attendu. Le passage de la première forme à la deuxième est appelé *lambda-lifting* :

Sucre syntaxique	Code équivalent

`cps int f(void)`
`{`
`  int x = 0;`
`  g();`
`  return x;`
 	
`  cps void g(void) {`
`  x = 3;`
`  }`
`}`
	`cps void g(int x) {`
	`  x = 3;`
	`}`
	
	`cps int f(void)`
	`{`
	`  int x = 0;`
	`  g(x);`
	`  return x;`
	`}`

Dans ce premier essai, on veut modifier directement la valeur de `x` : comme on le voit, on ne modifie que la variable propre à `g`, ce qui n'a aucun effet sur variable `x` de `f`. On voudrait que le code renvoie 3, mais on voit sur la version de droite qu'il renvoie 0.

Sucre syntaxique	Code équivalent

`cps int f(void)`
`{`
`  int x = 0;`
`  g();`
`  return x;`
 	
`  cps void g(void) {`
`  int *p = &x;`
`  *p = 3;`
`  }`
`}`
	`cps void g(int x) {`
	`  int *p = &x;`
	`  *p = 3;`
	`}`
	
	`cps int f(void)`
	`{`
	`  int x = 0;`
	`  g(x);`
	`  return x;`
	`}`

Dans ce deuxième essai, on a eu l'idée de passer par un pointeur, récupérant l'adresse de la variable `x`, pour ensuite la modifier, mais `x` est toujours une variable libre de la fonction interne, et celle-ci se réfère donc à une copie de `x`. Au final, c'est toujours la variable `x` propre à la fonction `g` qui est modifiée : le code renvoie toujours 0.

Fonction écrite	Fonction équivalente

`cps int f(void)`
`{`
`  int x = 0;`
`  int *p = &x;`
`  g();`
`  return x;`
 	
`  cps void g(void) {`
`  *p = 3;`
`  }`
`}`
	`cps void g(int *p) {`
	`  *p = 3;`
	`}`
	
	`cps int f(void)`
	`{`
	`  int x = 0;`
	`  int *p = &x;`
	`  g(p);`
	`  return x;`
	`}`

Cette solution fonctionne enfin : le code renvoie 3. Nous procédons comme dans un code C classique : afin de modifier la valeur d'une variable locale dans une autre fonction, il faut lui passer la copie d'un pointeur vers cette variable.

Fonctions CPS primitives. CPC fournit quelques primitives pour coopérer avec l'ordonnanceur. La primitive `cpc_yield` insère un point de coopération dans une fonction CPS ; `cpc_sleep` demande à l'ordonnanceur de n'exécuter la suite qu'au bout d'un certain temps ; `cpc_io_wait` demande à l'ordonnanceur d'attendre qu'un descripteur de fichier soit prêt avant d'effectuer la suite du calcul. On dispose aussi de variables de condition (`cpc_condvar` et fonctions associées[CK10]) pour la synchronisation des flots de contrôle.

1.2 Le compilateur CPC

La compilation d'un programme CPC s'effectue par une succession de transformations dont la validité est prouvable (et prouvée), aboutissant à un code C valide en style de programmation par passage de continuations.

Les principales passes :

- **un premier *lambda-lifting*** ayant pour effet de supprimer les fonctions (CPS) internes mentionnées ci-dessus.
- **une encapsulation des variables** dont l'adresse est demandée (`malloc` en début de fonction, et `free` en fin)
- **la mise sous forme CPS-convertible**, c'est à dire le découpage des fonctions CPS en de nouvelles sous-fonctions, définies à l'intérieur de la fonction mère.
- **un deuxième *lambda-lifting***, qui permet d'extraire les sous-fonctions nouvellement crées, et d'en faire des fonctions à part entière.
- **la conversion CPS**, qui finalise le travail, rajoutant la continuation comme arguments aux fonctions en ayant besoin, transforme les `cpc_spawn`, etc.

1.3 Description des passes de CPC

Premier *lambda-lifting*. Les fonctions internes décrites en 1.1, qu'elles soient introduites implicitement ou explicitement par l'utilisateur, sont extraites pour devenir des fonctions globales : leurs variables libres deviennent des arguments, et reçoivent la valeur de ces variables au moment de l'appel de la fonction, le C étant un langage passant les arguments des fonctions par valeur. Des exemples de lambda-lifting ont aussi été donnés en 1.1.

Encapsulation des variables. CPC copie ses variables d'une fonction à l'autre, évitant normalement la nécessité de les allouer. Cependant, si le programmeur est amené à prendre l'adresse d'une variable, la copie de cette variable n'a pas la même adresse que cette dernière : il est donc nécessaire de les encapsuler. Pour cela, CPC détecte chaque variable v dont l'adresse $\&v$ est demandée, lui alloue une zone mémoire m_v à l'aide de `malloc`, affecte la valeur v à l'emplacement mémoire m_v si v est un argument de la fonction, remplace chaque occurrence de v par le déréférencement $*m_v$ de m_v, et désalloue m_v avec `free` à la fin de la fonction. Au cas où la fonction retourne v, une sauvegarde préalable de la valeur de $*m_v$ est effectuée dans une variable temporaire, qui sera ensuite retournée.

Avant encapsulation	Après encapsulation

`cps void g(int *p) {`
`  *p = 3;`
`}`
	`cps void g(int *p) {`
	`  *p = 3;`
	`}`

`cps int f(void)`
`{`
`  int x = 0;`
 	
`  g(&x);`
 	
`  return x;`
`}`
	`cps int f(void)`
	`{`
	`  int *x =`
	`  (int*) malloc(sizeof(int));`
	`  int tmp;`
	`  *x = 0;`
	`  g(& (*x));`
	`  tmp = *x;`
	`  free(x);`
	`  return tmp;`
	`}`

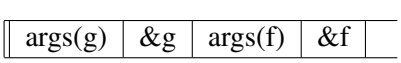
Mise sous forme CPS-convertible. La mise sous forme CPS-convertible consiste à transformer toutes les fonctions étiquetées par le mot-clé `cps`, qui sont donc destinées à devenir des fonctions CPS, en fonctions CPS-convertibles. Pour cela, CPC remplace les boucles par des étiquettes et des *gotos*, et parcourt le code jusqu'à ce qu'à ce qu'il ne soit plus sous forme CPS-convertible : à cet emplacement, il

ajoutera un *goto* suivi de son étiquette associée. Le code ainsi modifié est facilement CPS-convertible : chaque étiquette correspond au début d'une nouvelle fonction, et chaque *goto* pointant sur cette étiquette à un appel de cette fonction. Les fonctions sont simples à créer, car définies comme fonctions internes ; par ailleurs, tous leurs appels sont en position terminale.

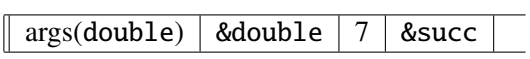
Deuxième *lambda-lifting*. Cette phase est en réalité identique au premier *lambda-lifting* (voir section 1.3) : les variables libres de chaque fonction interne sont ajoutées comme arguments de cette fonction, puis la fonction est définie comme globale. En revanche, la correction de cette transformation n'est pas évidente, et constitue une partie de la thèse de Kerneis [KC10]. Je vais m'efforcer ici d'en donner l'intuition, mais il faut bien comprendre que la démonstration formelle est autrement plus complexe. Lors de la conversion CPS, des affectations de variables ont pu être réalisés dans des fonctions internes, et doivent pourtant bien modifier les valeurs des variables de la fonction englobante, ce qui n'est pas possible étant donné que les variables sont passées par copie. Mais en réalité, comme dit au paragraphe précédent, les appels des fonctions internes sont *toujours* en position terminale : ainsi les variables de la fonction d'origine ne seront certes pas modifiés, mais on est certain qu'elles ne serviront pas après l'appel à une fonction interne, d'où la correction de ce *lambda-lifting*.

Conversion CPS. La passe de conversion CPS termine la compilation en créant les continuations, en les reliant aux fonctions CPS, en transformant les mots-clés en primitives C. Nous verrons ici ce qui me semble important à la bonne compréhension du projet, à savoir la gestion des continuations par rapport à l'appel et au retour de fonctions, et le traitement du `cpc_spawn`.

Les continuations CPC sont vraiment très semblables aux continuations décrites dans le préliminaire : ce sont des piles sur lesquelles se trouvent des pointeurs de fonctions, et leurs arguments. La structure d'une continuation peut se représenter ainsi :



Rappelons-nous l'exemple pris dans le préliminaire : `double( succ(7) )` (page 7). Comme nous l'avons vu, la continuation de `succ(7)` est constituée d'un appel à la fonction `double`. Avec la représentation de CPC, cela donnerait :



L'exécution de `succ(7)` ne pose pas de problème, mais comment pousser l'argument de `double` sur la continuation, puisqu'on ne connait pas encore sa valeur ? Le compilateur CPC étant capable de déterminer l'espace mémoire que prend l'argument de `double`, il laisse un emplacement mémoire libre : c'est la fonction `succ` qui écrira à cet emplacement sa valeur de retour.

2 E-CPC

2.1 Le compilateur E-CPC

2.1.1 Principe

Le compilateur E-CPC est une modification du compilateur CPC, compilant aussi un code CPC en un code C par passes successives, et qui produit un programme qui a exactement les mêmes effets que s'il était compilé par CPC. E-CPC introduit des environnements qui encapsulent l'ensemble des variables d'une fonction CPS : ainsi, les problèmes qui se posaient pour CPC lors du second *lambda-lifting* disparaissent naturellement, et il n'est plus nécessaire d'effectuer une encapsulation préalable de certaines valeurs.

2.1.2 Passes principales

Les principales passes utilisées dans E-CPC reprennent naturellement celles de CPC, puisque dans les deux cas il est nécessaire d'effectuer une conversion CPS. On retrouvera donc en particulier les phases de mise sous forme CPS-convertible et de conversion CPS. Le premier *lambda-lifting* est maintenu, car le langage CPC permet les fonctions internes. La mise sous forme CPS-convertible générant des fonctions internes, il est aussi nécessaire d'effectuer le second *lambda-lifting*, mais alors devenu trivial nous le verrons.

Les passes principales sont donc :

- **Un premier *lambda-lifting*** comme pour CPC.
- **Une phase de préparation** pour les environnements :
  - création d'un pointeur d'environnement,
  - positionnement des instructions de libération mémoire
  - gestion de la valeur de retour de la fonction.
- **La mise sous forme CPS-convertible**
- **La génération d'environnements**, c'est à dire :
  - création et l'allocation de la structure
  - affectation de ses champs
  - remplacement des variables par leurs indirections

  Le pointeur d'environnement est la seule variable restante dans les sous-fonctions.
- **Un deuxième *lambda-lifting***
- **La conversion CPS**

2.1.3 Description des passes et correction

On remarquera que dans le cadre de E-CPC, une la mise sous forme CPS-convertible réduit la CPS-convertibilité au cas où seulement une fonction CPS peut être appelée (plus éventuellement un appel à une fonction interne introduite par le

compilateur), en premier lieu pour des questions de simplifications relativement à un problème d'un cas particulier expliqué ci-après.

Phase de préparation. Après le premier *lambda-lifting*, il est nécessaire d'effectuer une phase préliminaire à la mise sous forme CPS-convertible afin que celui-ci se passe correctement. En effet, lors de l'introduction d'environnements, des instructions d'allocation et de libération de blocs mémoire pour ces environnements sont nécessaires. En particulier, les instructions de libération (« `free` ») sont placées à la fin des fonctions, et de ce fait, une fonction qui était CPS-convertible peut cesser de l'être. Par exemple, si g est une fonction CPS :

Fonction CPS-convertible	Fonction non-CPS-convertible

`cps int f() {`
`  int x = 0;`
`  return g(x);`
`}`
	`cps int f() {`
	`  struct f_env env;`
	`  int tmp;`
	`  env->x = 0;`
	`  tmp = g(env->x);`
	`  free(env);`
	`  return tmp;`
	`}`

Par ailleurs, une autre nécessité de cette phase est qu'après la mise sous forme CPS-convertible, il est presque impossible de discerner où la fonction d'origine termine : les « `return` » des fonctions créées ne correspondent évidemment pas nécessairement aux « `return` » de la fonction d'origine.

Dans CPC, les valeurs de retour des fonctions étaient écrites dans des « *trous* » mémoire dans la continuation, correspondants à un argument de la prochaine fonction à être exécutée. CPC choisit en effet pour toutes ses fonctions le même emplacement mémoire : la fonction appelante peut donc le « deviner ». Etant donné qu'E-CPC utilise des environnements, l'emplacement de la valeur de retour est différent pour chaque fonction appelée, aussi ai-je choisi pour E-CPC une autre approche que celle de CPC : la fonction appelante doit spécifier l'adresse de retour de la foncion appelée en argument de celle-ci. Ainsi, toutes les fonctions CPS retournant en un type « T » différent de « `void` » se voient pourvues d'un nouvel argument de type « $*T$ » et ne renvoient désormais plus rien ; leur appel est aussi modifié en conséquence. Si l'argument en question est le pointeur nul, le retour est ignoré. En voici un exemple :

Avant transformation	Après transformation

`cps int f(int x) {`
`  int tmp;`
`  tmp = x + 3;`
`  return tmp;`
`}`
	`cps void f(int *retval, int x) {`
	`  void *environnement;`
	`  int tmp;`
	`  tmp = x + 3;`
	`  if (retval != NULL) {`
	`  *retval = tmp;`
	`  }`
	`  free(environnement);`
	`  return;`
	`}`

`cps void g(void)`
`{`
`  int x;`
`  x = f(4);`
`  printf ("%d\n", x);`
`  return;`
`}`
	`cps void g(void)`
	`{`
	`  int x;`
	`  f(&x, 4);`
	`  printf ("%d\n", x);`
	`  return;`
	`}`

Notons qu'il est impossible d'allouer les environnements dès cette étape, car la mise sous forme CPS-convertible est susceptible d'introduire de nouvelles variables.

Mise sous forme CPS-convertible. E-CPC n'autorise pas les fonctions CPS-convertibles à comporter deux appels successifs à des fonctions CPS du code d'origine : il peut y en avoir deux, lorsqu'il a introduit la deuxième. En effet, à ce stade, les fonctions CPS ont déjà été modifiées par E-CPC comme décrit au précédent paragraphe, et en particulier toutes les fonctions CPS retournent « `void` ». Considérons le cas suivant :

```
cps void g(void);

cps void f(void)
{
  g();
  g();
  return;
}
```

On obtiendrait alors (avec `f_aux` la sous fonction générée automatiquement dans le cas de E-CPC) :

CPC	E-CPC

`cps void f(void)`
`{`
`  g();`
`  g();`
`  return;`
`}`
	`cps void f(void)`
	`{`
	`  g();`
	`  f_aux();`
	`  return;`
	
	`  cps void f_aux(void) {`
	`  g();`
	`  return;`
	`  }`
	`}`

Afin de bien comprendre pourquoi cette modification est nécessaire, il faut comprendre comment CPC gère les continuations, ce qui est décrit en 1.3. Considérons le code suivant, qui reste inchangé jusqu'à la mise sous forme CPS-convertible dans le cas de CPC, et sa modification par E-CPC.

avant mise sous forme CPS-convertible	
CPC	E-CPC

`{`
`  ...`
`  x = f();`
`  g(x);`
`  ...`
`}`
	`{`
	`  ...`
	`  f(&x);`
	`  g(x);`
	`  ...`
	`}`

La mise sous forme CPS-convertible de CPC coupe après `x = f()`, mais elle ne le ferait pas après `f(&x)` puisque `f` ne retourne pas de valeur : si E-CPC ne forçait pas la coupure ici, c'est la valeur de `x` *avant* l'appel à `f` qui serait passée à `g`. E-CPC coupe deux instructions successives de la même manière que CPC, en ajoutant un nouveau goto suivi de son étiquette associée. Le reste de la mise sous forme CPS-convertible est rigoureusement le même pour E-CPC que pour CPC, et le résultat du code précédent est :

après mise sous forme CPS-convertible	
CPC	E-CPC

`{`
`  ...`
`  x = f();`
`  __aux(x);`
`  return;`
 	
`  cps void __aux(int x) {`
`  g(x);`
`  ...`
`  }`
`}`
	`{`
	`  ...`
	`  f(&x);`
	`  __aux();`
	`  return;`
	
	`  cps void __aux(void) {`
	`  g(x);`
	`  ...`
	`  }`
	`}`

Génération d'environnements. C'est à dire la création et l'allocation de la structure, ainsi que l'affectation de ses champs, et le remplacement des variables par leurs indirections.

C'est la deuxième et dernière partie de la gestion d'environnements. E-CPC commence par récupérer toutes les variables locales, qui sont supprimées, et les arguments de la fonction (hormis le pointeur d'environnement créé lors de la première partie), afin de pouvoir créer la structure correspondante, puis génère les instructions d'allocation mémoire et d'initialisation des champs correspondants aux arguments de la fonction. Il est très important de souligner que ces instructions sont placées *au tout début* de la fonction traitée, car celle-ci reste ainsi CPS-convertible. Après cela, il suffit de remplacer chaque variable (variables locales et arguments) par une référence à sa valeur dans l'environnement.

Chaque sous-fonction créée par la mise sous forme CPS-convertible ne prend aucun argument. Anticipant la passe de *lambda-lifiting*, et disposant des informations nécessaires, E-CPC peut facilement affecter à chaque sous-fonction son unique et immuable argument : l'environnement.

Passes suivantes. Le 2e lambda-lifting est maintenant trivial : en effet, toutes les variables des fonctions internes sont liées (ou globales), et ces fonctions peuvent donc être extraites sans autre précaution ; la conversion CPS quant à elle n'est pas modifiée.

3 Résultats expérimentaux

E-CPC, un compilateur fonctionnel et correct

Le compilateur E-CPC actuellement implémenté fonctionne totalement, en cela qu'il compile un code CPC en un code natif ayant les mêmes résultats que s'il avait été compilé par le compilateur CPC : il a été testé sur tous les tests créés pour CPC, et donne les mêmes résultats.

Les comparaisons des deux compilateurs indiquent que les conjectures de Chroboczek et Kerneis se vérifient, à savoir que CPC est plus efficace qu'E-CPC. Cependant ce dernier reste tout à fait utilisable et ses performances sont à quelques pourcents celles de CPC. Trois jeux de tests ont permis de mesurer les efficacités des différents compilateurs : le premier mesure les performances de primitives particulières, et les compare avec d'autres librairies pour la concurrence (section 3.1) ; le deuxième mesure la rapidité d'exécution d'un programme (section 3.2) ; le troisième mesure le temps de réponse de serveurs web (section 3.3).

3.1 Comparatifs de primitives

Le premier jeu de tests reprend ceux mis en place par Chroboczek sur des programmes simples qu'il a écris, permettant de comparer l'efficacité de différentes primitives entre CPC et d'autres systèmes de gestion des *threads* (nptl, pth et st) ; j'y ai donc intégré E-CPC. Quatre caractéristiques sont ici représentées :

- *call et cps-call*, le coût d'un appel de fonction CPS par rapport à un coût d'appel de fonction native.
- *switch*, le changement de contexte, c'est à dire le passage d'un *thread* à l'autre.
- *spawn*, la création d'un nouveau *thread*.
- *condvar*, les variables de condition, permettant de synchroniser les *threads* entre eux.

Ce jeu de tests a été exécuté sur plusieurs machines différentes, dont trois sont ici représentées, car elles ont des caractéristiques différentes, et les résultats s'en ressentent. J'ai rassemblé dans un tableau le quotient des temps d'exécutions pris entre CPC et E-CPC afin de pouvoir les comparer facilement, cependant nous le verrons, si ces résultats montrent que CPC est plus rapide que E-CPC, relire les valeurs sur les diagrammes permet de voir que le compilateur E-CPC crée tout de même des programmes rapides.

E-CPC / CPC	cps-call	switch	spawn	condvar
Core 2 Duo	2,45	1,67	2,18	1,13
Pentium M	2,35	1,75	3,12	1,08
MIPS-32 4KEc	2,92	1,43	1,59	0,91

Autant CPC et E-CPC sont très rapides sur les primitives de concurrence (*switch*, *spawn*, *condvar*), autant ils sont lents sur les appels de fonctions CPS, ce qui est

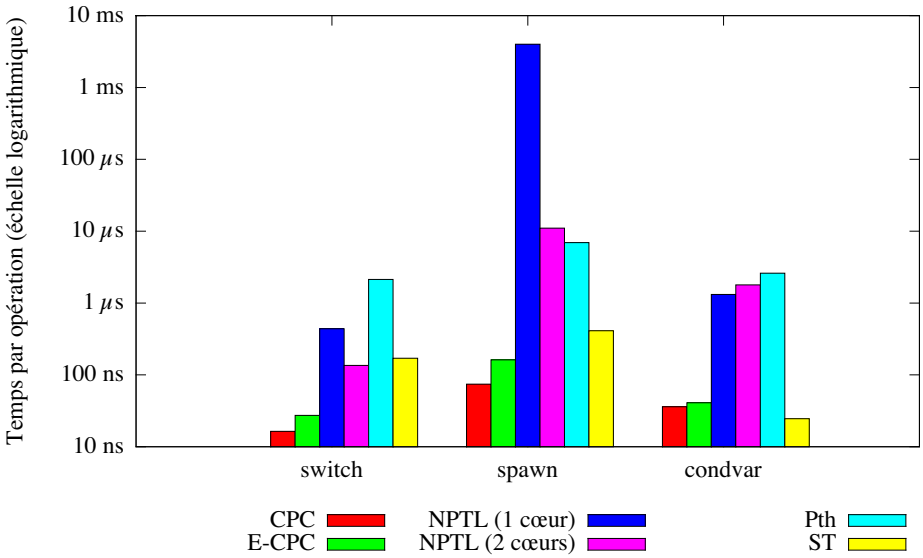
particulièrement vrai avec E-CPC dans son implémentation actuelle. Les appels de fonctions CPS étant omniprésents dans les programmes écrits en langage CPC, on peut penser que c'est le point essentiel à optimiser dans E-CPC, et qu'il explique en partie les performances des tests suivants.

En ce qui concerne le changement de contexte, CPC est un peu plus rapide que E-CPC (le facteur variant entre 1,43 et 1,75), sans grande différence entre les architectures.

La création de nouveaux processus est beaucoup moins régulière : sur Pentium M, CPC va jusqu'à 3,12 fois plus vite que E-CPC, alors que sur MIPS-32 4KEc, CPC n'est qu'à un facteur 1,59 de E-CPC.

La gestion des variables de condition offre des résultats surprenants aussi : CPC est plus rapide que E-CPC sur le Core 2 Duo, presque identique sur Pentium M, et moins rapide (facteur 0,91) sur MIPS-32 4KEc. Si dans l'ensemble E-CPC rattrape son retard sur MIPS-32 4KEc, les résultats sont en revanche hétérogènes sur les deux autres machines : le Pentium M est plus rapide pour CPC sur les instructions de changement de contexte et de naissance de threads, mais moins rapide pour les variables de condition que le Core 2 Duo.

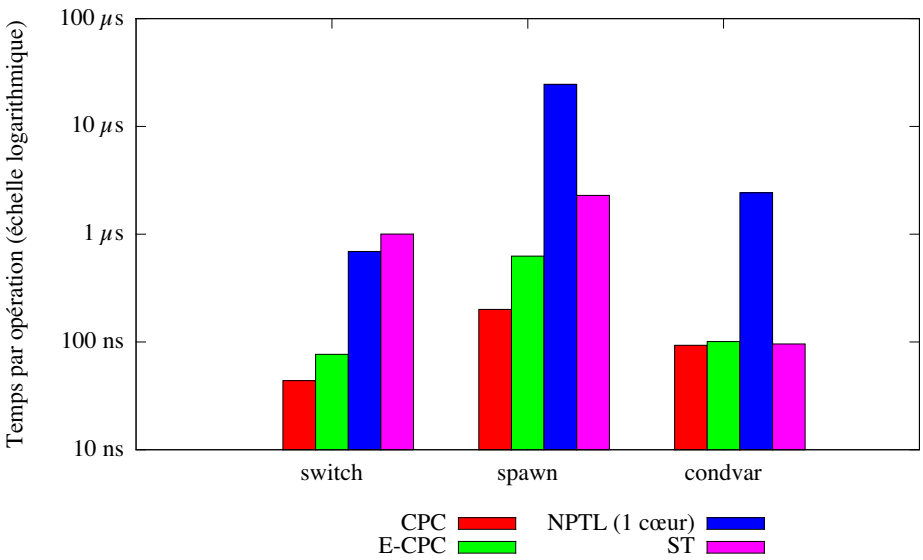
Les figures 1, 2 et 3 sont intéressantes et complémentaires de ces résultats : elles montrent que si CPC est presque toujours plus rapide que E-CPC, ce dernier reste aussi compétitif que CPC par rapport aux autres librairies C de gestion des *threads* : les mêmes conclusions peuvent être tirées pour CPC et E-CPC.



programme	call	cps-call	switch	spawn	condvar
CPC	0	20	16	74	36
E-CPC	0	49	27	162	41
NPTL (1 cœur)	0	–	439	4×10^6	1318
NPTL (2 cœur)	0	–	135	11037	1791
Pth	0	–	2130	6940	2602
ST	0	–	170	411	25

Architecture x86-64, processeur CPU Intel Core 2 Duo cadencé à 3,17 GHz.
Noyau linux 2.6.39

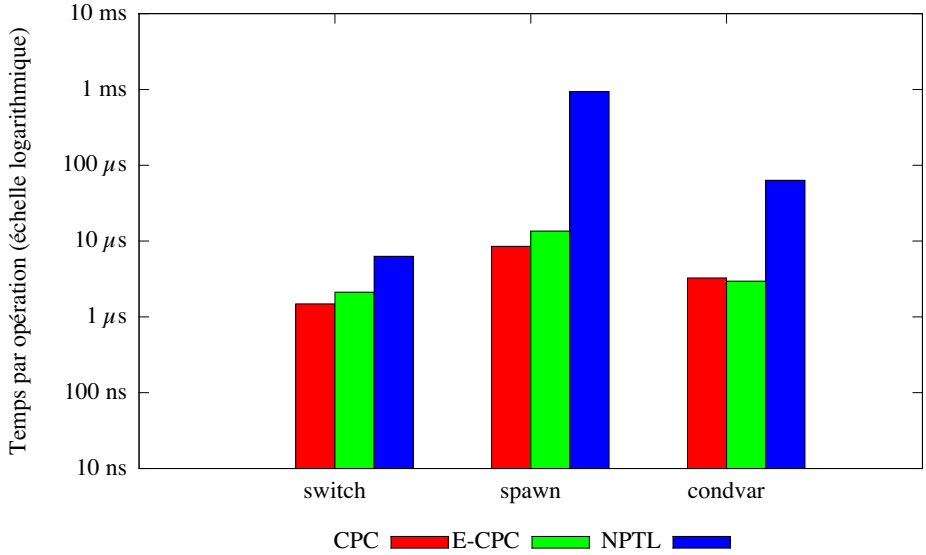
FIGURE 1 – résultats des tests des primitives sur Intel Core 2 Duo



programme	call	cps-call	switch	spawn	condvar
CPC	2	51	44	201	93
E-CPC	2	120	77	626	101
NPTL (1 cœur)	2	–	691	24 575	2 426
ST	2	–	1 003	2 293	96

Architecture x86-32, processeur Intel Pentium M cadencé à 1,7 GHz
Noyau Linux 2.6.39 ; sched_compat_yield activé

FIGURE 2 – Résultats des tests sur Pentium M



programme	call	cps-call	switch	spawn	condvar
CPC	5	2 018	1 482	8 519	3 268
E-CPC	5	5 888	2 119	13 544	2 962
NPTL	5	–	6 310	933 689	63 305

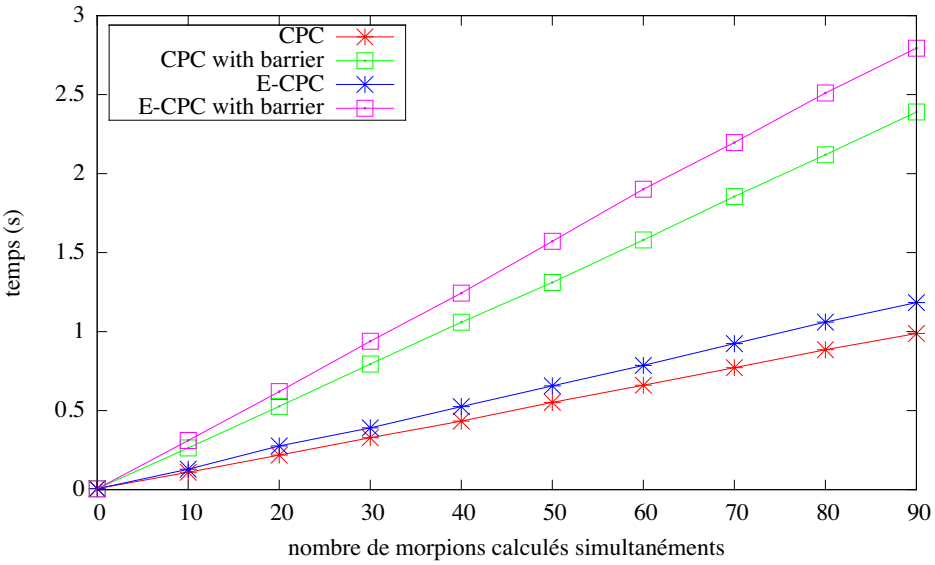
Architecture MIPS, processeur MIPS-32 4KEc cadencé à 184 MHz
Noyau OpenWrt, ulibc, 2.6.37.6 ; schd_compat_yield activé

FIGURE 3 – Résultats des tests sur MIPS-32 4KEc

3.2 Tests sur un programme

Les tests ci-dessus mesurent la performance d'une primitive donnée, mais pour considérer vraiment leur influence dans un programme, il faudrait en connaître la répartition. Dans le souci de comparer les deux compilateurs, CPC et E-CPC, j'ai donc écrit un programme calculant toutes les positions possibles d'un morpion (sans considération cependant qu'il faut qu'il y ait un nombre de croix étant le même à 1 près du nombre de ronds — il y a donc 3^9 réponses). Il n'utilise que des *threads* CPC : il n'y a en particulier aucun *thread* détaché.

La grille est remplie case par case, et à chaque case que l'on doit remplir, on crée deux nouveaux *threads*. Ainsi, nous avons 3 *threads*, chacun ayant une copie de la grille précédente jusqu'à la nouvelle case à remplir, et recevant une valeur différente pour cette case. Les deux implémentations diffèrent en cela que dans la première, les nouvelles grilles sont allouées manuellement avec `malloc` et désallouées avec `free`, alors que dans la deuxième, on utilise des primitives de synchronisation de *threads* de la librairie CPC appelées *barrières* : elles permettent d'attendre que *n* *threads* soient arrivées à un point donné avant de permettre l'exécution de la suite du code ; cela permet ici d'écrire son code en utilisant des tableaux comme définis



	CPC	E-CPC	E-CPC / CPC
Normal	0.01101	0.01318	1.20
Barrière	0.02646	0.03130	1.18

Calcul des coefficients de régression linéaire

Tests réalisés sur un Dell Inspiron 9300, processeur Intel Centrino 1,87 Ghz, cadencé à 800 MHz

FIGURE 4 – résultats des tests pour un programme (deux implémentations)

sur la pile. Il faut préciser que dans mon programme, l'utilisation des barrières est très fréquente.

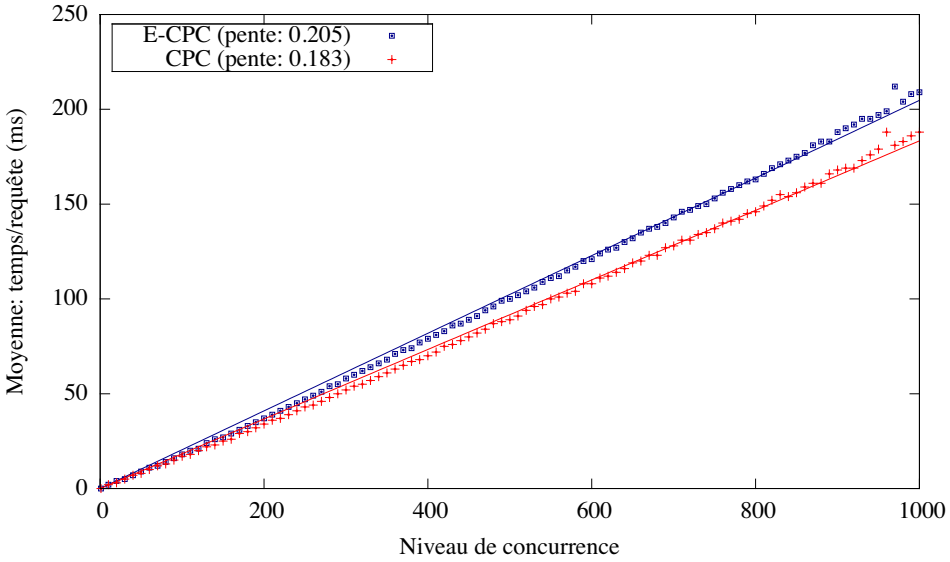
Il s'avère ici (figure 4) que CPC était facteur $1,2$ plus rapide qu'E-CPC. En particulier nous pouvons observer que les deux implémentations donnent lieu à une croissance linéaire. De plus, si les barrières ont un coût important, l'utilisation ou non de ces primitives n'influe pas sur la comparaison des performances des deux compilateurs.

3.3 Tests sur des serveurs web

Le dernier test a été de reprendre des serveurs web minimalistes que Kerneis avait écrits [KC09] : encore une fois j'ai réécrit les scripts pour qu'ils soient à même de comparer les deux compilateurs.

Un serveur compilé avec CPC et E-CPC est exécuté dans ses deux versions sur une première machine, puis des requêtes sont envoyées par une autre machine à la première via le réseau. Le *niveau de concurrence* pour un serveur est le nombre de requêtes auxquelles il doit répondre en même temps : on la mesure en fait depuis le client, qui maintient donc un certain nombre de requêtes en vol. Par exemple,

un niveau de concurrence de 100 signifie que le client maintient 100 requêtes en vol. Les résultats représentent ici le temps moyen de traitement des requêtes pour des niveaux de concurrence différents. Le résultat est assez proche de celui observé pour le morpion : CPC est 1, 12 fois plus rapide que E-CPC.



CPC	E-CPC	E-CPC / CPC
0.183	0.205	1.12

Calcul des coefficients de régression linéaire

Serveur : IBM Thinkpad, Intel Pentium M 1.7 GHz, cadencé à 600 MHz.
CPU presque saturé pendant les tests.
Client : Nec workstation, Intel Core 2 Duo 3.16 GHz en mode performance.
CPU à 30% pendant les tests.

FIGURE 5 – résultats des tests sur des serveurs

Conclusion

La réalisation du compilateur E-CPC a permis de montrer que l'utilisation d'environnements n'est pas aussi coûteuse qu'on aurait pu le croire. De plus, bien que la version actuelle d'E-CPC remplisse son rôle pour les besoins de cette étude, elle demeure relativement naïve, et peut être améliorée.

Tout d'abord, on peut penser qu'allouer les environnements toujours à des emplacements mémoire proches voire similaires pourrait augmenter la réactivité du programme, en cela que le cache serait mieux utilisé : c'est possible en allouant l'environnement sur la continuation, en utilisant une primitive proche de `cpc_alloc` (utilisée dans CPC) à la place de `malloc`, et de même une primitive proche de `cpc_dealloc` au lieu de `free`.

Un deuxième intéret d'allouer les environnements sur la continuation est que les fonctions internes introduites par le compilateur puis *lambda-liftées* n'auraient même plus besoin de prendre l'adresse de l'environnement en paramètre : connaissant la taille de l'environnement, qui est décidée à la compilation, et remarquant que leur environnement est au sommet de la continuation, elles pourraient aisément retrouver son adresse.

Une autre optimisation possible serait d'effectuer une analyse de vivacité fine des variables (CPC et E-CPC effectuent déjà une analyse grossière dont je n'ai pas parlé pour simplifier les explications), afin de réduire le nombre de variables utilisées dans l'environnement.

Certaines passes introduisent des éléments qui sont inutiles à E-CPC, comme par exemple de rajouter systématiquement une variable qui récupère le résultat d'une fonction CPS ayant une valeur de retour (s'il n'y en a déjà une).

Enfin, on pourrait probablement garder plusieurs appels à des fonctions CPS pour des fonctions CPS-convertibles en affinant la coupure ; mais attention à cette « optimisation » : certes le code produit serait moins lourd, et moins de fonctions CPS seraient crées, donc en pratique moins d'appels à ces fonctions seraient faits, ce qui gagne en rapidité, mais pousser plusieurs fonctions signifie aussi allouer plusieurs environnements en même temps, ce qui peut représenter un coût en mémoire plus important, problème très limité dans le cas de CPC (seuls les arguments sont conservés).

Il serait intéressant d'effectuer ces améliorations afin de voir l'évolution des performances par rapport à la version actuelle de E-CPC, et par rapport à CPC lui-même.

Références

[AC09] Pejman Attar and Yoann Canal. Réalisation d'un *seeder* bittorrent en CPC, June 2009.

[AHT+02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack

management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley, March 1988.

[CK10] Juliusz Chroboczek and Gabriel Kerneis. *The CPC manual*, 2010.

[KC09] Gabriel Kerneis and Juliusz Chroboczek. Are events fast ? Technical report, PPS, Université Paris 7, January 2009.

[KC10] Gabriel Kerneis and Juliusz Chroboczek. Continuation-Passing C, compiling threads to events through continuations. Submitted for publication, 2010.