Continuation-Passing C :

Transformations de programmes pour compiler la concurrence dans un langage impératif

Gabriel Kerneis

sous la direction de Juliusz Chroboczek

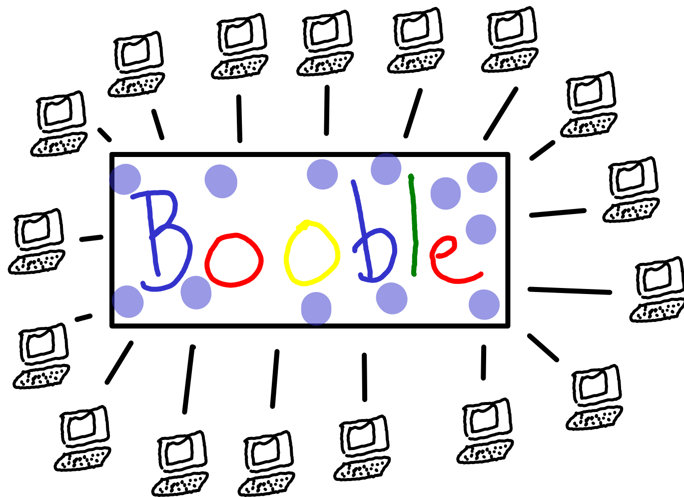
Laboratoire PPS, Université Paris Diderot

Thèse de doctorat, Paris, 9 novembre 2012

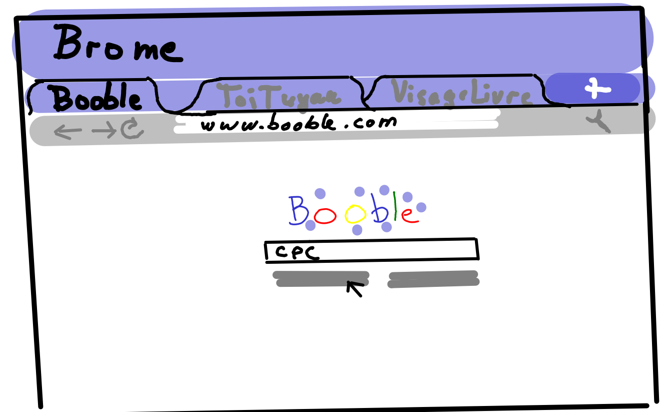
Concurrence

Programme **concurrent** : exécute plusieurs tâches simultanément.

Serveur web



Navigateur web



Objectif

Écrire un programme **hautement** concurrent,
aussi **facilement** que possible.

Comptons des moutons

```
void count(char *animal) {
  int n = 0;
  while(1) {
    printf("%d %s\n", n++, animal);
    sleep(1);
  }
}
count("sheep");
```

Séquentiel = une tâche à la fois

Comptons des poissons aussi

```
void count(char *animal) {
  int n = 0;
  while(1) {
    printf("%d %s\n", n++, animal);
    sleep(1);
  }
}
count("sheep");
count("fish");
```

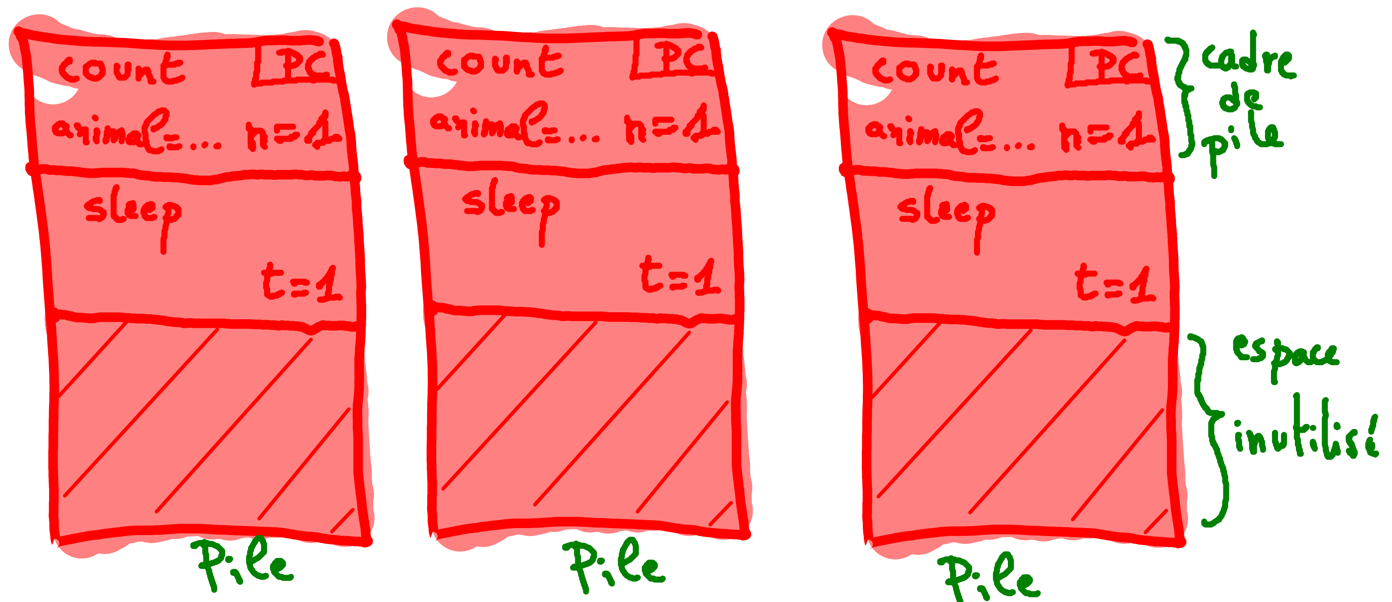
Ce programme n’est pas concurrent.

Comptons en chœur

```
void count(char *animal) {
  int n = 0;
  while(1) {
    printf("%d %s\n", n++, animal);
    sleep(1);
  }
}
pthread_create(..., count, "sheep");
pthread_create(..., count, "fish");
```

Anatomie d'un thread

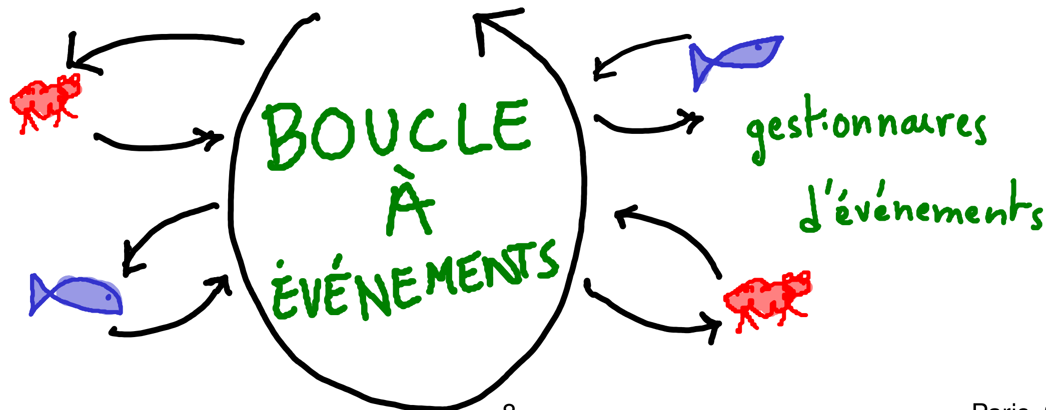
Chaque thread posséde sa propre **pile**,
qui capture l'état courant du thread.



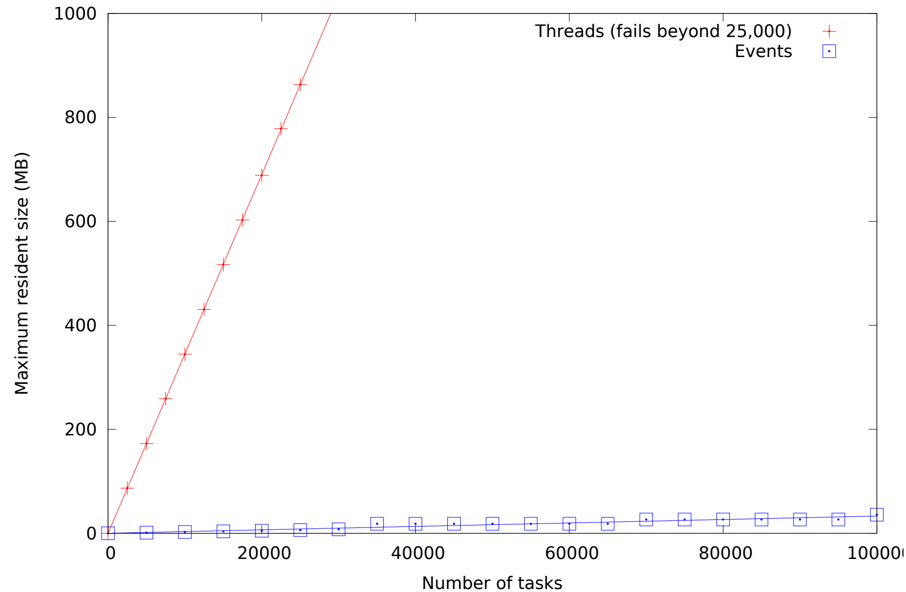
Allouer et maintenir une pile est coûteux.

Événements

```
struct state { int n; char *animal; };
void count(struct state *s) {
  printf("%d %s\n", s->n++, s->animal);
  runAfter(1, &count, s);
}
count(&((struct state){ 0, "sheep"}));
count(&((struct state){ 0, "fish"}));
startEventLoop();
```



Utilisation mémoire



Chaque thread occupe un espace fixé.

Les événements sont bien plus légers.

Les événements sont difficiles

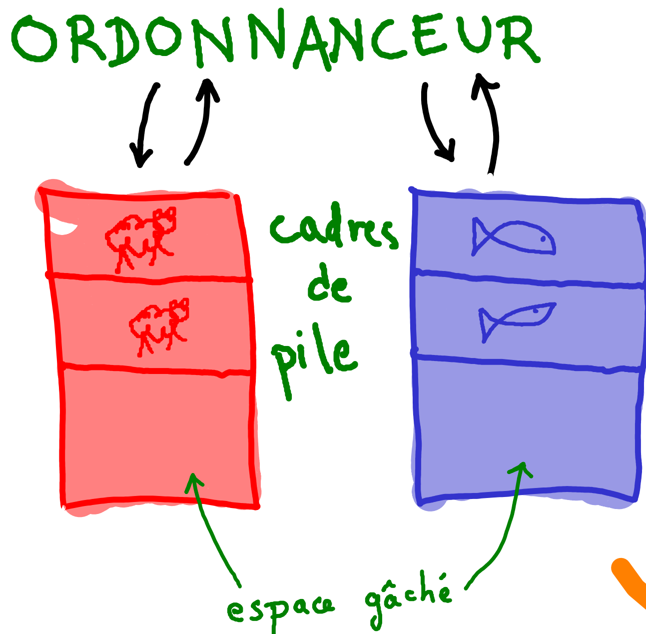
Il faut tout faire soi-même.

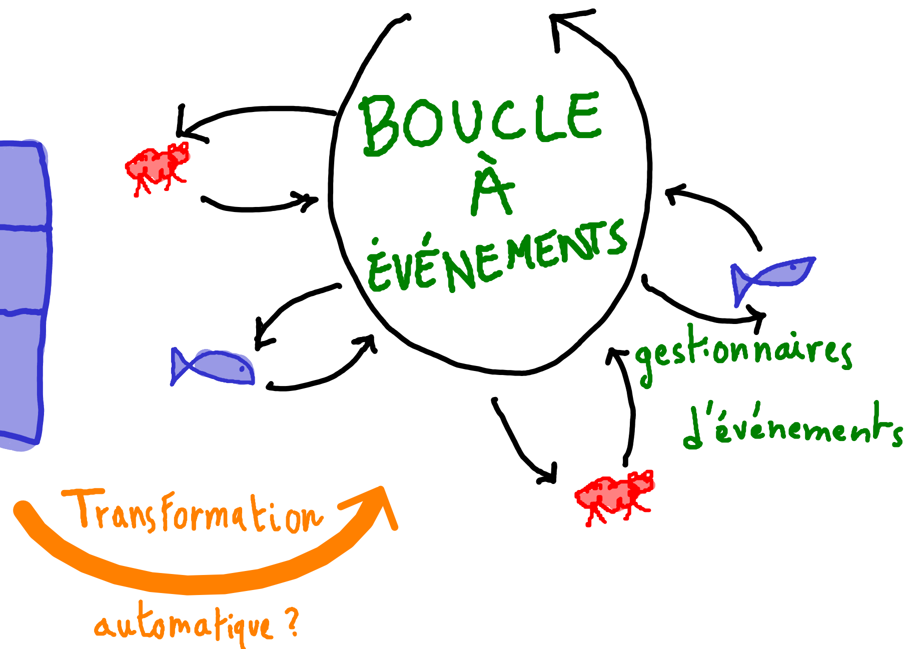
Difficiles à lire	Flot de contrôle rompu
Difficiles à écrire	Flot de données manuel
Difficiles à déboguer	Absence de pile
Difficiles à paralléliser	Un seul cœur par défaut
Souvent insuffisants	Interfaces bloquantes

Des threads aux événements

Threads
lisibles, lourds

Événements
difficiles, légers





Thèse

Les programmes concurrents

écrits dans un style à **threads**

sont **traduisibles** automatiquement,

par une suite de

transformations source-source **prouvées**,

en programmes à **événements**

équivalents et **efficaces**.

Plan

Le langage CPC

Technique de compilation

Preuve de correction

Résultats expérimentaux

Conclusions

Le langage CPC

Continuation-Passing C

Extension du langage C
pour l'écriture de programmes
hautement concurrents.

Threads légers unifiés,
implémentés par des événements
ou exécutés par des threads natifs.

Retournons à nos moutons

```
cps void count(char *animal) {
  int n = 0;
  while(1) {
    printf("%d %s\n", n++, animal);
    cpc_sleep(1);
  }
}
cpc_spawn count("sheep");
cpc_spawn count("fish");
```

Threads CPC **coopératifs** (par défaut) : ne peuvent être interrompus qu'aux appels de primitives CPS.

Code hybride

Threads coopératifs : agréables mais **limités**.

On peut pas se passer des threads natifs pour :
– les appels systèmes bloquants,
– les bibliothèques bloquantes,
– le parallélisme.

Threads CPC : coopératifs par défaut, convertibles **dynamiquement** en threads natifs (et inversement).

Continuation-Passing C (en détail)

Extension du C avec des threads légers

Mots-clés

- `cpc_spawn` créer un thread
- `cps` écrire une fonction coopérative

Primitives

- `cpc_yield` passer la main
- `cpc_sleep` dormir un peu
- `cpc_wait` synchroniser des threads
- `cpc_io_wait` attendre des données
- `cpc_link` alterner entre coopératif et natif

Technique de compilation

des threads aux événements

Architecture générale

Threads (CPC)

Traducteur CPC

Événements (C)

Compilateur C ← **Runtime (C)**

Exécutable

Traducteur CPC

Programme CPC

0. Boxing limité (5 %)

1. Explicitation du flot de contrôle

2. Élimination des gotos

3. Lambda lifting

4. Conversion CPS

Programme C

Un exemple : le compte-à-rebours

```
cps void countdown(int n) {
  while(n > 0) {
    printf("%d\n", n--);
    cpc_sleep(1);
  }
  printf("time is over!\n");
}
```

1. Explicitation du flot de contrôle

Objectif général

Sauvegarder le flot de contrôle aux points de coopération.

Explicitation du flot de contrôle

Boucles remplacées par des **gotos** (sauts) et des branchements.

Théoriquement trivial, implémentation délicate *(switch, break, continue, changements minimaux)*.

1. Explicitation du flot de contrôle

```
cps void countdown(int n) {
  loop:
    if(n <= 0) goto timeout;
    printf("%d\n", n--);
    cpc_sleep(1);
    goto loop;
  timeout:
    printf("time is over!\n");
}
```

2. Élimination des gotos

Les **gotos** et les **étiquettes** ne peuvent pas être sauvegardés tels quels :

Conversion des **gotos** en **appels terminaux** à des fonctions internes :

```
goto k;
```

$\downarrow$

```
k(); return;
```

(van Wijngaarden 1966 – Algol 60)

Indéfinie en C

2. Élimination des gotos

```
cps void countdown(int n) {
  cps void loop() {
    if(n <= 0) { timeout(); return; }
    printf("%d\n", n--);
    cpc_sleep(1); loop(); return;
  }
  cps void timeout() {
    printf("time is over!\n"); return;
  }
  loop(); return;
}
```

3. Lambda lifting

Les fonctions internes et leurs variables libres ne sont pas définies en C :

Copier les **variables libres**
en paramètre des **fonctions internes**
(*parameter lifting*).

Puis extraire ces fonctions closes (*block floating*).

(Johnsson 1985 – Lazy ML)

Pas correct en C

3. Lambda lifting

```
cps void loop(int n) {
  if(n <= 0) { timeout(); return; }
  printf("%d\n", n--);
  cpc_sleep(1); loop(n); return;
}
cps void timeout() {
  printf("time is over!\n"); return;
}
cps void countdown(int n) {
  loop(n); return;
}
```

4. Conversion CPS

Continuation : représentation de
« la suite du calcul ».

Conversion CPS : transforme les appels de fonctions en appels terminaux à des fonctions recevant une continuation.

```
sleep(1); print(n); return;
```

$$\downarrow$$

```
sleep⋆(1, λk.print⋆(n, k)); return;
```

(Plotkin 1975 – λ-calcul)

Pas correcte en C

4. Conversion CPS

```
cps void loop(int n, cont *k) {
  if(n <= 0) { timeout(k); return; }
  printf("%d\n", n--);
  cpc_sleep(1, push(loop, n, k)); return;
}
cps void timeout(cont *k) {
  printf("time is over!\n");
  invoke(k); return;
}
cps void countdown(int n, cont *k) {
  loop(n, k); return;
}
```

0. Boxing

Boxing = ajout d'une indirection aux variables.

```
int x = 4; int *p = &x;
```

```
int *x_b = malloc(sizeof(int));
*x_b = 4; p = x_b;
```

Boxer toutes les variables locales serait correct, mais coûteux *(indirection, cache)*.

Compromis

CPC ne boxe **que** (5 %) les variables **extrudées** (ie. celles dont l'adresse est retenue par `&`).

Traducteur CPC (bilan)

Programme CPC

0. Boxing limité (5 %)

1. Explicitation du flot de contrôle

2. Élimination des gotos

3. Lambda lifting

4. Conversion CPS

Programme C

Preuve de correction

Lambda lifting et CPS conversion
dans un langage impératif

Lambda lifting en C

Problème : copier une variable, modifier l'originale.

```
void f() {
  int x = 1;
  g() { x = 2; };
  g(); printf("%d\n", x);
}
```

est transformé en

```
void f() {
  int x = 1;
  g(int x) { x = 2; };
  g(x); printf("%d\n", x);
}
```

Correction du lambda lifting

Théorème : dans un langage impératif
en appel par valeur sans variable extrudée,
le *parameter lifting* est correct pour les variables libres des fonctions dont toutes les fonctions internes sont appelées en position terminale.

Idée : on ne revient jamais observer l'original.

Corollaire : le lambda lifting effectué par le traducteur CPC est correct.

Preuve : les fonctions lambda-liftées sont issues de la conversion des gotos, et les variables extrudées sont encapsulées.

Conversion CPS en C

Problème : copier une variable, modifier l'originale.

```
f(); g(x); return;
```

est transformé en

```
push(g, x, k); f(k); return;
```

Si `f` peut modifier `x`, l'**évaluation anticipée** de la variable est incorrecte.

Correction de la conversion CPS

Théorème : dans un langage impératif en appel par valeur sans variable extrudée, statique ou globale, l'évaluation anticipée est correcte pour les fonctions closes.

Corollaire : la conversion CPS effectuée par le traducteur CPC est correcte.

Preuve : les fonctions CPS-converties sont closes par lambda lifting, et les variables extrudées sont encapsulées.

Preuves de correction

Langage impératif réduit à l'affectation, la conditionnelle et les fonctions internes. Réduction naïve avec *store* et environnements.

Réduction « optimisée » équivalente à la réduction naïve, pour la preuve de correction du lambda lifting.
(environnements minimaux, discipline de pile)

Introduction de **continuations abstraites** pour la preuve de correction de la conversion CPS *(sur lesquelles agissent **push** et **invoke**)*.

Résultats expérimentaux

Efficacité et utilisabilité

CPC est presque **aussi rapide** que les implémentations de threads **les plus rapides**, tout en permettant la création d'au moins **5 fois plus** de threads.

CPC est **utilisable** par des **étudiants** de master, pour écrire un **serveur** réseau **complet**.

Efficacité

Trois ensembles de **mesures expérimentales** :

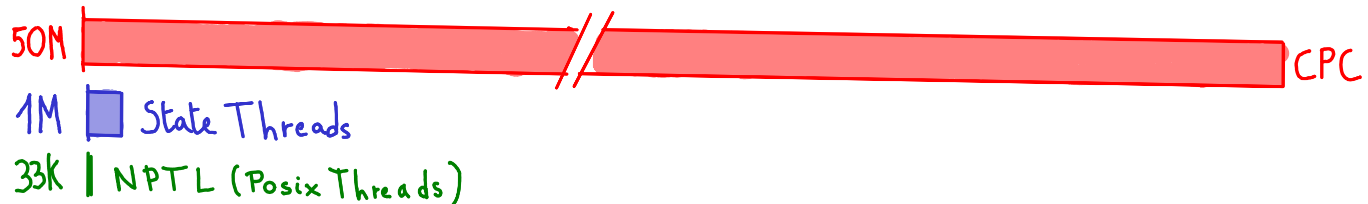
Occupation mémoire

Rapidité

- Primitives
- Serveurs réseaux

Occupation mémoire

Les threads CPC sont très légers.



Nombre de threads possibles avec 4 Go de RAM

Les continuations sont étendues dynamiquement au besoin.

Vitesse des primitives

Bibliothèque	*call*	*CPS-call*	*switch*	*cond*	*spawn*
nptl	2		691	2 426	24 575
ST	2		1 003	96	2 293
CPC	2	54	46	91	205
eCPC	2	120	77	101	626

Temps en nanosecondes
Intel Pentium M (32 bits) à 1.7 GHz, Linux 2.6.38

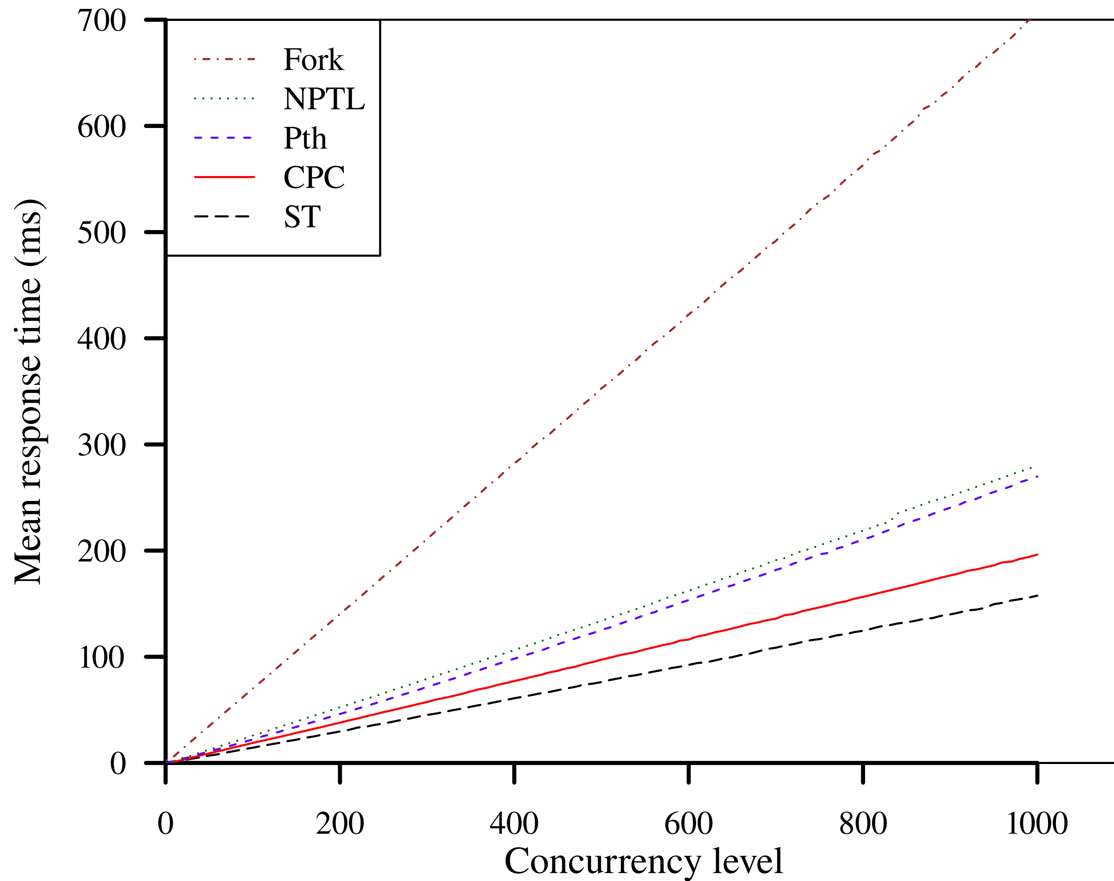
Primitives très efficaces.

Mais au prix d'**appels CPS** plus lents.

eCPC : boxing plus lent que le lambda-lifting

(Boutier, Kerneis, 2012)

Serveurs web minimalistes



Hekate, un seeder BitTorrent

Écrit par 2 **étudiants de master** *(Attar, Canal)*
2 000 lignes de CPC et 1 000 lignes de C.

Code hybride : entrées-sorties bloquantes,
bibliothèque externe *(curl)*

Machine de bureau : 1 000 clients,
2 % du processeur, sature le lien réseau.

Système embarqué : 100 % du processeur,
saturé par les entrées-sorties USB.

Conclusions

Contributions

Implémentation complète et **utilisable** du langage CPC.

Technique de **compilation prouvée** :
– lambda lifting dans un langage impératif,
– conversion CPS dans un langage impératif.

Utilisabilité et **efficacité** :
– implémentation d'Hekate, un serveur BitTorrent,
– mesures expérimentales montrant que CPC est
aussi ou plus rapide que les autres implémentations,
et permet la création d'au moins 5 fois plus de threads.

Implémentation alternative, eCPC,
utilisant du boxing au lieu du lambda lifting,
et évaluation du coût associé.

Pistes futures

Analyse de vie des variables dans le λ-lifting

Boucles à événements multiples

Variables de condition en mode natif

Entrées-sorties asynchrones (Windows)

(travail en cours par Boutier)

Réimplémentation en LLVM, ou en CompCert

Adaptation à Javascript

Conclusion

Les programmes concurrents

écrits dans un style à **threads**

sont **traduisibles** automatiquement,

par une suite de

transformations source-source **prouvées**,

en programmes à **événements**

équivalents et **efficaces**.