HirondML
Fair thread migration in O'Caml

Emmanuel Chailloux (PPS) - Julien Verlaguet (Univ P6)
http://www.pps.jussieu.fr/˜emmanuel/Public/Dev/HirondML

HLPP 2005

Motivations

- Fair Threads in OCaml
  - An alternative to POSIX threads
  - Native implementation
- Fair Thread Migration
  - A high level contruction for distributed programming
  - Permits communication in a type-safe manner
- Objectives :
  - A clear semantic
  - An efficient implementation

Fair Threads

- Frédéric Boussinot
- MIMOSA EMP-CMA project / Inria Sophia Antipolis
- http://www-sop.inria.fr/mimosa/rp
- The existing implementations
  - C
  - Java
  - Scheme

Fair Threads : Caracteristics

- Mixing cooperative and preemptive policies
  - Each thread is attached to a scheduler
  - Threads within a same scheduler are cooperative
  - Schedulers are concurrent to each-other
- The synchronisation system
  - Based on “instants”
  - Takes place inside a scheduler
  - Each operation is “finite”
    $\Rightarrow$ No dead-lock

Inside a scheduler : Cooperative policy

Instant :

- Execution of each thread until the next cooperation point
- The scheduling policy is of type “round-robin”

The different ways to cooperate :

- explicitly : function `cooperate`
- implicitly : waiting `await`

FT : Exemple1

```
let sched=Fthread.create_scheduler();;

let rec fth x=
  Printf.printf "I am ft number %d\n" x;
  Fthread.cooperate();
  fth x
;;

Fthread.create sched fth 1;
Fthread.create sched fth 2;
Fthread.start_scheduler sched;
Fthread.exit();;

(* Output : *)
(* I am ft number 1
   I am ft number 2
   I am ft number 1
   ... *)
```

Synchronisation

- A signal is emitted during an “instant”
- A signal reaches all the fair threads waiting during an instant
- A thread can only wait for a signal during a limited number of instants

Exemple 2

```
let fth1()=
  Printf.printf "I am %d I am waiting \n" x;
  await signal;
;;
```

FT : Caracteristics for continuation migration

- Cooperative threads : clear semantic
  - No mutex
  - No lost signals
  - No distributed signals
- Schedulers : grouped migration ?
  - A scheduler is a safe regroupement in terms of synchronisation
- How to implement thread migration ?
  - A possible answer : continuations

Continuations in OCaml

- CPS implementation ?
  - Heavy modifications to the compiler
  - Less efficient in our case
- Stack copy implementation ?
  - How do we translate code pointers ?
  - How do we rebind data ?
  - What do we copy ?

$\Rightarrow$ a new semantic

Migrating FT : Conditions

- Compatible computers (same architecture)
- Same program for all the computers involved
- A computer identification mechanism
- A scheduler dealing with migration
- Evaluation of all the global variables

FT migration : semantic

- Migration of a FT from a `src` to a `dst` with its local environment and its execution context
  - All the accessible references from the local environment are copied (from `src` to `dst`)
- The global variables are relinked
- The FT is attached to the main scheduler

Exemple : Chat

```
let dest=132;; (* distant computer *)
let home=Migrate.addr_comp;; (* source computer

let rec loop h =
  Printf.printf "Enter your msg\n";
  flush stdout;
  let s = read_line() in
    Migrate.migrate dest ;
    Printf.printf "%s\n" s ; flush stdout;
    Migrate.migrate h;
    loop h;;

loop home;;
Migrate.exit();;
```

Migration in OCaml

- The OCaml marshaller
  - Generic and polymorphic
  - Works on functional values
- The Garbage Collector
  - Efficient stack scanning
  - Efficient heap exploration

Migration : Implementation (1)

- Capture of local variables
- Capture of the execution context
- Detachement of the thread system
- Marshalling

Migration : Implementation (2)

- Unmarshalling
- Update of the local environment
  - Stack allocation
  - Insertion of the copied values
- Update of the execution context
  - New stack pointer
- Attachement to the thread system

Efficient migration ?

- Risk of heap absorption
- Liveness analysis (in native)
- The programmer is in charge of the local environment

Exemple (1)

```
let master_addr=0;;
let job_list=ref [];;

let master home=
  let n=ref 0 in

  let rec job_producer()=
    job_list :=
      (fun x -> x + !n) :: !job_list;
    Migrate.cooperate();
    job_producer() in

  job_producer()
;;
```

Exemple (2)

```
let rec slave home=
  migrate master_addr;
  let job=
    match !job_list with
        [] -> fun _ -> raise Not_found
     | j :: rl -> j in
  Migrate.migrate home;
  (try
    printf "Result is %d\n" (job home)
  with Not_found ->
    printf "No job available\n");
  Migrate.cooperate();
  slave home;;
```

Exemple (3)

```
create master master_addr;;

for i=1 to 3 do
  for j=0 to 5 do
    create slave i
  done
done;;

Migrate.exit();;
```

Migration on different architectures ? (1)

- Theoratically possible, technically difficult
- The difficulties :
  - Different code pointers
  - Different data representations
  - Different optimisations
  - Exception handling
  - Different execution contexts

Migration on different architectures ? (2)

- A bytecode version, advantages :
  - Code pointers problem solved
  - Regular optimisations
- A bytecode version, drawbacks :
  - No liveness analysis
  - Doesn’t solve all the problems :
    - Interpretor execution context
    - Exceptions

Conclusion

- A semantic lead by the implementation
- Some other languages are more expressive : Acute, ...
- A simple semantic
- Possibly efficient