Permission Based Verification of Dara Race Freeness for Lock Free Programs

Christian Haack [1] Clément Hurlin [2]

University of Nijmegen [1]

Inria Sophia-Antipolis [2]

ParSec

Parallelism and Security

June 20, 2007

Introduction

Writing correct multithreaded programs software is:

1. *difficult* (Flanagan and Qadeer).
2. *notoriously difficult* (Jacobs et al.).
3. *notoriously tricky* (Peyton Jones et al.).

Introduction

Writing correct multithreaded programs software is:

1. *difficult* (Flanagan and Qadeer).
2. *notoriously difficult* (Jacobs et al.).
3. *notoriously tricky* (Peyton Jones et al.).

- Difficulties arise when objects are shared.
- Read/write or write/write conflicts: data race.

Permissions Against Races

Boyland's *Checking interferences with fractional permissions*:

- Associate each location with a permission.
- Full permission 1 permits to read and write.
- Split permissions $\frac{1}{2}, \frac{1}{4}, \ldots$ permit only to read.

Permissions Against Races

Boyland's *Checking interferences with fractional permissions*:

- Associate each location with a permission.
- Full permission 1 permits to read and write.
- Split permissions $\frac{1}{2}, \frac{1}{4}, \ldots$ permit only to read.

Our goal:

- Lift this to a Java-like language.
- To handle lock free algorithms (possibly with arrays).
  - Proof obligations related to array index arithmetic.
  - We will delegate them to a theorem prover in an implementation.

Permissions Against Races

π	$::=$		permissions
	1		full permission (needed for writing)
	$\texttt{split}(\pi)$		split permission (needed for reading)
	α		permission variable

Permissions Against Races

π	$::=$	permissions
	1	full permission (needed for writing)
	$\mathtt{split}(\pi)$	split permission (needed for reading)
	α	permission variable

P, Q, R	$::=$	permissions formulas
	e	(Boolean) expression
	$\mathtt{Perm}(r[\kappa], \pi)$	reference r has permission π to $r.\kappa$
	$\mathtt{Final}(r[\kappa])$	full permission to $r.\kappa$ is lost forever
	$\mathtt{fa}(x; e; P)$	for all x, e implies P
	$P * Q$	permission-splitting conjunction

- κ is a field or $*$.
- r is a `final` reference path.

Permissions Against Races

Requirements/assumptions are written as methods pre/postconditions:

```
class DoubleTwoRows extends Thread{
  int a[][];
  int row;

  //@ requires Perm(a[row..row+1][*],1);
  //@ ensures  Perm(a[row..row+1][*],1);
  void run(){
    int j = 0;
    while(j < a.length){
      //@ loop_invariant Perm(a[row..row+1][*],1);
      a[row][j] = 2*a[row][j];
      a[row+1][j] = 2*a[row+1][j];
      j++;
    }
  }
}
```

Permission Splitting/Combination

Permission splitting is *not* idempotent:

$$\texttt{Perm}(r[\kappa], \pi) \not\equiv \texttt{Perm}(r[\kappa], \pi) * \texttt{Perm}(r[\kappa], \pi)$$

Permission Splitting/Combination

Permission splitting is *not* idempotent:

$$\texttt{Perm}(r[\kappa], \pi) \not\equiv \texttt{Perm}(r[\kappa], \pi) * \texttt{Perm}(r[\kappa], \pi)$$

However, non `Final` permissions can be split into two *smaller* permissions:

$$\texttt{Perm}(r[\kappa], \pi) \equiv \texttt{Perm}(r[\kappa], \texttt{split}(\pi)) * \texttt{Perm}(r[\kappa], \texttt{split}(\pi))$$

Permission Splitting/Combination

- `Final`$(r[\kappa])$ means that $r[\kappa]$ is readonly forever.
- `Final` permissions can be split an infinite number of times but cannot be recombined to a full permission:

$$\texttt{Final}(r[\kappa]) \equiv \texttt{Final}(r[\kappa]) * \texttt{Final}(r[\kappa])$$

- This extends `Java`'s `final`: fields can be finalized at any point (not only during constructor).

Permission Splitting/Combination

Splitting an array into different parts:

$$\texttt{Perm}(r[*],\pi) \equiv \texttt{fa}(x;\, 0 \texttt{<=} x \,\&\, x \texttt{<} r.\texttt{length};\, \texttt{Perm}(r[x],\pi))$$

$$\texttt{!}e \texttt{ | } \texttt{!}e' \Rightarrow \texttt{fa}(x\,;\, e \texttt{ | } e'\,;\, P) \equiv \texttt{fa}(x;e;P) * \texttt{fa}(x;e';P)$$

Taking Advantage of Aliasing

Reference equality is built into the logic:

$$e == e' * P[e/x] \equiv e == e' * P[e'/x]$$

- This allows to verify more programs since permissions can “flow” from one alias to another.
- Boyland used alias types for the same purpose.

Taking Advantage of Aliasing

Reference equality is built into the logic:

$$e == e' * P[e/x] \equiv e == e' * P[e'/x]$$

- This allows to verify more programs since permissions can “flow” from one alias to another.
- Boyland used alias types for the same purpose.

$$\frac{\Gamma \vdash v : \Gamma(\ell) \quad \Gamma; P \vdash Q \quad \ell \notin Q \quad \Gamma; \{\, Q * \ell == v\} \vdash c : T\{R\}}{\Gamma; \{P\} \vdash \ell = v; c : T\{R\}} \text{ (Var Set)}$$

Taking Advantage of Aliasing

Reference equality is built into the logic:

$$e == e' * P[e/x] \equiv e == e' * P[e'/x]$$

- This allows to verify more programs since permissions can "flow" from one alias to another.
- Boyland used alias types for the same purpose.

$$\frac{\Gamma \vdash v : \Gamma(\ell) \quad \Gamma; P \vdash Q \quad \ell \notin Q \quad \Gamma; \{\, Q * \ell == v\} \vdash c : T\{R\}}{\Gamma; \{P\} \vdash \ell = v; c : T\{R\}} \text{(Var Set)}$$

$$\frac{C <: \Gamma(\ell) \quad \Gamma; P \vdash Q \quad \ell \notin Q \quad \Gamma; \{\, Q * \texttt{Perm}(\ell[*], 1)\} \vdash c : T\{R\}}{\Gamma; \{P\} \vdash \ell = \texttt{new}\ C; c : T\{R\}} \text{(New)}$$

Taking Advantage of Aliasing


```
class C{
  int x;
  int y;

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],1);
  void m(){ ... }

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],split(1));
  void n(){ ... }
}
```

```
void main(){
  C c = new C();
    {Perm(c[*],1)}



}
```

Taking Advantage of Aliasing


```
class C{
  int x;
  int y;

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],1);
  void m(){ ... }

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],split(1));
  void n(){ ... }
}
```

```
void main(){
  C c = new C();
    {Perm(c[*],1)}

  C a = c;
    {a == c * Perm(c[*],1)}

}
```

Taking Advantage of Aliasing


```
class C{
  int x;
  int y;

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],1);
  void m(){ ... }

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],split(1));
  void n(){ ... }
}
```

```
void main(){
  C c = new C();
    {Perm(c[*],1)}

  C a = c;
    {a == c * Perm(c[*],1)}

  a.m();
    {a == c * Perm(c[*],1)}


}
```

Taking Advantage of Aliasing

```
class C{
  int x;
  int y;

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],1);
  void m(){ ... }

  //@ requires Perm(this[*],1);
  //@ ensures  Perm(this[*],split(1));
  void n(){ ... }
}
```

```
void main(){
  C c = new C();
    {Perm(c[*],1)}

  C a = c;
    {a == c * Perm(c[*],1)}

  a.m();
    {a == c * Perm(c[*],1)}

  c.n();
    {a == c * Perm(c[*],split(1)}
}
```

Fork/Join Patterns

`fork`, `run`, and `join` are particular methods:

- `t.fork()` spawns a new thread `t` and calls `t`'s `run` method.
- `t.join()` returns if `t` is a terminated thread (i.e. `t`'s `run` method is finished)

Fork/Join Patterns

`fork`, `run`, and `join` are particular methods:

- `t.fork()` spawns a new thread `t` and calls `t`'s `run` method.
- `t.join()` returns if `t` is a terminated thread (i.e. `t`'s `run` method is finished)
- Our system uses `run`'s precondition as the precondition for `fork`.
- Our system uses `run`'s postcondition as the postcondition for `join` under additional conditions:

Fork/Join Patterns

`fork`, `run`, and `join` are particular methods:

- `t.fork()` spawns a new thread `t` and calls `t`'s `run` method.
- `t.join()` returns if `t` is a terminated thread (i.e. `t`'s `run` method is finished)
- Our system uses `run`'s precondition as the precondition for `fork`.
- Our system uses `run`'s postcondition as the postcondition for `join` under additional conditions:

`join` permission:

- $\texttt{Perm}(r[\texttt{join}], \texttt{split}^n(1))$: Reference r has permission to use $\frac{1}{2^n}$-th of r.`join`'s post-condition.
- $\texttt{Perm}(r[\texttt{join}], \texttt{split}^n(1)) \equiv \texttt{Perm}(r[\texttt{join}], \underbrace{\texttt{split}(\ldots(\texttt{split}}_{n\ \texttt{split}}(1)\ldots)))$

Fork/Join Patterns


```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}
```

```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
}
```

Fork/Join Patterns


```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
    {Perm(s[*],1)  *  Perm(cm1[join],1)}
  ...
  Cloner cm8 = new Cloner(s);

}
```

Fork/Join Patterns


```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);
    {Perm(s[*],1) * Perm(cm1[join],1)
     * ... * Perm(cm8[join],1)}



}
```

Fork/Join Patterns


```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

    {Perm(s[*],1)}
  cm1.fork();

}
```

Fork/Join Patterns

```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

    {Perm(s[*],1/2)  *  Perm(s[*],1/2)}
  cm1.fork();
    {Perm(s[*],1/2)}

}
```

Fork/Join Patterns

```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

  cm1.fork();
  ...
    {Perm(s[*],1/128)}
  cm8.fork();



}
```

Fork/Join Patterns

```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

  cm1.fork();
  ...
    {Perm(s[*],1/256) * Perm(s[*],1/256)}
  cm8.fork();
    {Perm(s[*],1/256)}



}
```

Fork/Join Patterns


```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

  cm1.fork();
  ...
  cm8.fork();

    {Perm(s[*],1/256) * Perm(cm8[join],1)}
  cm8.join();
    {Perm(s[*],1/128)}

}
```

Fork/Join Patterns


```
class Cloner{
  Subject s;

  ...

  //@ requires Perm(s[*],α);
  //@ ensures  Perm(s[*],α);
  void run(){ ... }
}
```

```
void main(){
  Subject s = new Subject();
    {Perm(s[*],1)}

  Cloner cm1 = new Cloner(s);
  ...
  Cloner cm8 = new Cloner(s);

  cm1.fork();
  ...
  cm8.fork();


  cm8.join();
  ...
    {Perm(s[*],1/2) * Perm(cm1[join],1)}
  cm1.join();
    {Perm(s[*],1)}
}
```

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

$$\frac{[\![r]\!]_s^h = o \qquad [\![\pi]\!] \leq \mathscr{P}(o,\kappa)(\mathit{ref}(r)_s^{h,t})}{\Gamma \vdash \mathscr{P}; h; s \models_t \mathtt{Perm}(r[\kappa],\pi)} \text{ (Valid Perm)}$$

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

$$\frac{[\![r]\!]_s^h = o \qquad [\![\pi]\!] \leq \mathscr{P}(o,\kappa)(\mathit{ref}(r)_s^{h,t})}{\Gamma \vdash \mathscr{P}; h; s \models_t \mathtt{Perm}(r[\kappa],\pi)} \text{ (Valid Perm)}$$

Permissions tables are defined such that:

$$\sum_{[\![r]\!]_s^h = o} \mathscr{P}(o,\kappa)(r) \leq 1$$

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

$$\frac{[\![r]\!]_s^h = o \qquad [\![\pi]\!] \leq \mathscr{P}(o,\kappa)(ref(r)_s^{h,t})}{\Gamma \vdash \mathscr{P}; h; s \models_t \texttt{Perm}(r[\kappa],\pi)} \text{ (Valid Perm)}$$

Permissions tables are defined such that:

$$\sum_{[\![r]\!]_s^h = o} \mathscr{P}(o,\kappa)(r) \leq 1$$

1. Two threads writing to location ℓ need permission 1 to ℓ.

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

$$\frac{[\![r]\!]_s^h = o \qquad [\![\pi]\!] \leq \mathscr{P}(o,\kappa)(ref(r)_s^{h,t})}{\Gamma \vdash \mathscr{P}; h; s \models_t \texttt{Perm}(r[\kappa], \pi)} \text{ (Valid Perm)}$$

Permissions tables are defined such that:

$$\sum_{[\![r]\!]_s^h = o} \mathscr{P}(o,\kappa)(r) \leq 1$$

1. Two threads writing to location ℓ need permission 1 to ℓ.
2. The sum of permissions to a location is less or equal than 1 in verified programs.

Semantics of Permission Formulas

Permission formulas are interpreted w.r.t. permission tables.

- $\Gamma \vdash \mathscr{P}; h; s \models_t P$
  - "P holds in permission table $\mathscr{P}$, heap h and local store s of thread t"

$$\frac{[\![r]\!]_s^h = o \qquad [\![\pi]\!] \leq \mathscr{P}(o,\kappa)(ref(r)_s^{h,t})}{\Gamma \vdash \mathscr{P}; h; s \models_t \texttt{Perm}(r[\kappa], \pi)} \text{ (Valid Perm)}$$

Permissions tables are defined such that:

$$\sum_{[\![r]\!]_s^h = o} \mathscr{P}(o,\kappa)(r) \leq 1$$

1. Two threads writing to location ℓ need permission 1 to ℓ.
2. The sum of permissions to a location is less or equal than 1 in verified programs.
3. Thus, verified programs do not contain data races.

Conclusion

Work in progress:

- Soundness.

Future work:

- Algorithmic checking ?
- Implementation ?
- More general system (locking).
- Alternative approach to avoid the `final` limitation (with `modifies` clause).
- Relationship with separation logic ?