Programming Languages

Dr. Michael Petter, Raphaela Palenta WS 2017/18

Exercise Sheet 5

Assignment 5.1 Restricted Transactional Memory

Consider the following code fragment on a machine with RTM and Caches:

```
int data = 0;
int s=0;
```

thread P_0:

```
while (s!=-1)
  if((s=_xbegin())==-1){
    data++;
    _xend();
  }
```

thread P_1:

```
if (_xbegin()==-1){
  data++;
  _xend();
}else {
  data++;
}
```

1. Fill in the gap with either "**will**", "**will not**" or "**may or may not**":

   After P_0 and P_1 both terminate, `data` **may or may not** evaluate to 1.

   Reason: The fail branch of P_1 is unprotected and may race the TM instruction of P_0

2. Fill in the gap with either "**will**", "**will not**" or "**may or may not**":

   After P_0 and P_1 both terminate, `data` **will not** evaluate to 3.

   Reason: P_0 will leave s evaluating to -1 as soon, as its transaction terminates successfully, thus P_0 will terminate the loop after P_0 has incremented data exactly once and commited. P_1 also only increments data once.

3. Consider the following interleaving of paths through the program:

   P_0 `s!=-1` `(s=_xbegin())==-1` `data++;` `s!=-1` `(s=_xbegin())==-1` `data++` `_xend()`
   P_1 `_xbegin()==-1` `data++;`
   $\longrightarrow$ time

   Draw a happened-before diagram of this interleaving. The initial cache states for **`s`** and **`data`** are $S0, S0$. (No store buffer and invalidate queue.)

4. Fix the program, such that 2 is the only value, that **data** may evaluate to after termination of both threads. (Of course without hardcoding!)

Suggested Solution 5.1

as follows:

```
int data = 0;
int s=0;
int mutex=1;
```

thread P_0:

```
while (s!=-1)

  if((s=_xbegin())==-1){
    if (!mutex>0) _xabort();
    data++;

    _xend();

  }
```

thread P_1:

```
if (_xbegin()==-1){

  data++;

  _xend();

}else {
  wait(mutex);
  data++;
  signal(mutex);
}
```

Assignment 5.2 STM vs. RTM

This time, we want to compare Tranactional Memory implementations with each other as well as the old implementations from tutorial sheet 3. Thus, we will equip the bumper allocator with TM implementations.
Equip the bumper allocation implementation with

- explicit RTM (this will only run on a CPU with transactional memory)
- GCC transaction extensions

Suggested Solution 5.2

1. with explicit RTM instructions: see `bumperrtm.c`
2. with gcc-extensions: see `bumpergcc.c`
3. with HLE (this is a bonus and was not asked in the assignment): see `bumperhle.c`