Frontend and Compiler Representations of Reducers for Clarity and Optimization

William S. Moses

wmoses@mit.edu

OpenCilk Meeting
October 9, 2020

CSAIL

Reductions

- Parallelizing code is often difficult as it inadvertently may create race conditions on variables.

- One class of these races include variables whose results need to be accumulated for later, but whose “current value” doesn’t need to be accessed in a parallel task

```
int total = 0;

//Parallelizable excluding the race on total
for(int i=0; i<n; i++) {
 // Don’t need value of total, just update
 total += array[i];
}

// Retrieve value after-the-fact
return total;
```

Cilk Reducers

- Cilk provides reducer hyperobjects that allow programmers to express an associative “blind update computation” in a race-free way.
- Ensures final result maintains ordering, if required (e.g. for a linked-list).
- Negligible theoretical overhead, built into runtime system.
- Large practical overhead, both in performance and linguistics.

```
void identity(void* reducer, void* value) {
  *((int*) value) = 0;
}
void reduce(void* reducer, void* left, void* right){
  *((int*) left) += *((int*) right);
}

CILK_C_DECLARE_REDUCER(int) total =
  CILK_C_INIT_REDUCER(int, reduce, identity, __cilkrts_hyperobject_noop_destroy);

REDUCER_VIEW(total) = 0;

CILK_C_REGISTER_REDUCER(total);

cilk_for(int i=0; i<n; i++) {
  REDUCER_VIEW(total) += array[i];

CILK_C_UNREGISTER_REDUCER(total);

return REDUCER_VIEW(total);
```

Proposed Syntax

```
void identity(void* reducer, int* value) {
  *value = 0;
}
void reduce(void* reducer, void* left, void* right) {
  *left += *right;
}

int __attribute__((reducer(reduce, identity)))
  total = 0;

//Parallelizable excluding the race on total
cilk_for(int i=0; i<n; i++) {
 // Don't need value of total, just update
 total += array[i];
}

// Retrieve value after-the-fact
return total;
```

Proposed Syntax

- Consider reducer variables as modifications to the type

```
int __attribute__((reducer(reduce, identity, destroy, allocate, deallocate)))
```

Advantages:

- Use of variable is equivalent to serial case
- Potential type-checking
- Serial Projection == removing attribute (default behavior)

Proposed Syntax

- Initialization (upon creation/allocation):
  - Allocate reducer object
  - Initialize current view corresponding to default constructor (C++) or default initialization (C)
- Use:
  - All LValue uses [e.g. references to underlying object] are replaced by the current reducer view
- Destruction (when leaving scope):
  - Destruct all (including current) reducer views
  - Deallocate reducer object
- Exceptions: TBD

Custom Types

```
struct Node{
   struct Node *next, *prev;
   int value;
};
typedef struct Node* List;

void list_construct(List* l){ *l = NULL; }

List list_add(List l, int x){
   List m = malloc(sizeof(*m));
   m->value = x;
   if(l){
      l->next->prev = m;
      m->next = l->next;
      l->next = m;
      m->prev = l;
   } else
      m->next = m->prev = m;
   return m;
}
```

```
//Reducer function
void list_merge(void* red, List* a,
List* b){
   List tmp = (*a)->next;
   (*a)->next = *b;
   tmp->prev = (*b)->prev;
   (*b)->prev->next = tmp;
   (*b)->prev = *a;
}

List __attribute__((reducer(list_merge,
list_construct, list_erase)))
  x = list_construct();

cilk_for (int i = 1; i < 10; i++)
     x = list_add(x, i);
```

Function Calls

```
void add2(int* value) {
  cilk_for(int i=0; i<3; i++) {
   // This races. We passed a pointer to an integer, the current
   // view. From this location in code there is no indication of
   // anything special to handle races.
   *value++;
  }
}

int __attribute__((reducer(reduce, identity))) total = 0;

cilk_for(int i=0; i<n; i++) {
  // Passes the current view, not the reducer
  add2(&total);
}
```

Reducerof

- Reducerof takes a reducer “LValue” and instead returns a pointer to the reducer object

- This is how reducers should be passed to subfunction calls which themselves are parallel

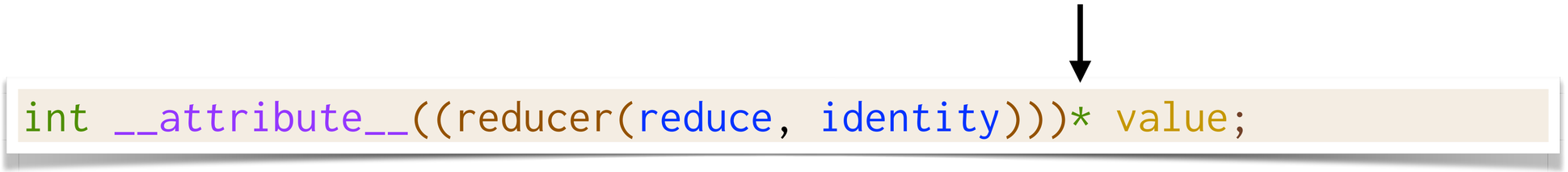
```
void add2(int __attribute__((reducer(reduce, identity)))* value) {
  cilk_for(int i=0; i<3; i++) {
   // This is safe.
   *value++;
  }
}

int __attribute__((reducer(reduce, identity))) total = 0;

cilk_for(int i=0; i<n; i++) {
  // Passes the current view, not the reducer
  add2(reducerof(total));
}
```

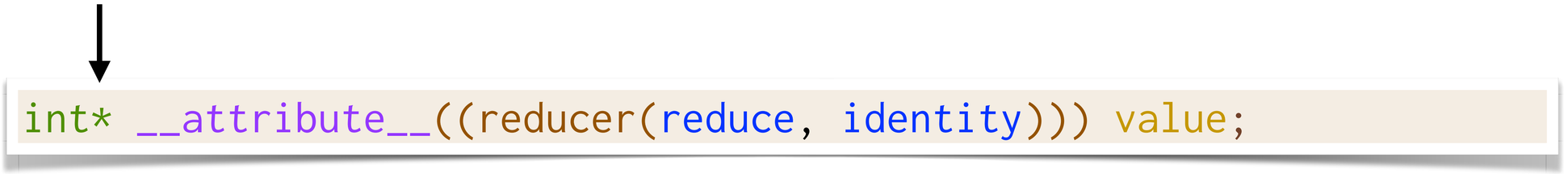
Type Representations

- Pointer to an integer reducer



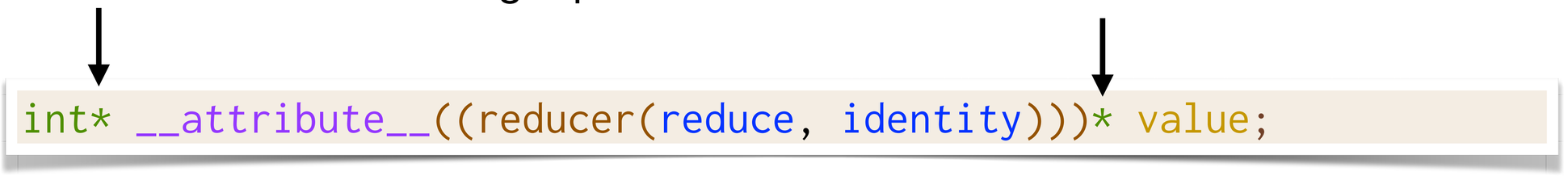
```
int __attribute__((reducer(reduce, identity)))* value;
```

- Reducer of an integer pointer



```
int* __attribute__((reducer(reduce, identity))) value;
```

- Pointer to a reducer of an integer pointer



```
int* __attribute__((reducer(reduce, identity)))* value;
```

Reducer Pointers & Arrays

- Heap-based reduces require a different mechanism of allocation

```
int __attribute__((reducer(reduce, identity)))* value =
   reducer_alloc(sizeof(int), reduce, identity);
```

- “Arrays of reducers” are not permitted

```
int __attribute__((reducer(reduce, identity))) value[3];
// Ambiguous sizing problems
//   (void*)value[1] - (void*)value[0] == sizeof(int) [per enclosed]
//   (void*)value[1] - (void*)value[0] == sizeof(reducer) [actual allocation]
// Both need to be true (for indexing/allocation, respectively)
```

- “Arrays of reducers pointers” ARE permitted

```
int __attribute__((reducer(reduce, identity)))* value[1] = {
   reducer_alloc(sizeof(int), reduce, identity), … };
```

Status & Limitations

- Syntax Prototype in Tapir @ LLVM version 8
- Reducer functions must be compile-time constants (and not dynamic)
  - Stems from part of type
  - Possible to remove restriction if forgo type checking subtype
- No reducer arrays, but instead reducer pointer arrays

Compiler Representation of Reducers

- Reducers are highly unoptimized inside the compiler

```
cilk_for(int i=0; i<n; i++)
  // separate lookup every iteration
  REDUCER_VIEW(total)++;

return REDUCER_VIEW(total);
```

```
cilk_for(int i=0; i<n; i+=10)
  for(int j=0; i<min(i+10,n); j++)
    // 10 x extraneous lookups
    REDUCER_VIEW(total)++;

return REDUCER_VIEW(total);
```

Reducers act as a barrier for existing optimizations

```
int foo(Matrix* MyMatrix) {
  total = …;
  cilk_for(int i=0; i<n; i++) {
    if (MyMatrix[i,0])
      REDUCER_VIEW(total) += MyMatrix->size();
  }
  return REDUCER_VIEW(total);
}
```

ReducerView reads/writes global memory which could alias MyMatrix

```
int foo(Matrix* MyMatrix) {
  total = …;
  int licm = MyMatrix->size();
  cilk_for(int i=0; i<n; i++) {
    if (MyMatrix[i,0])
      REDUCER_VIEW(total) += licm;
  }
  return REDUCER_VIEW(total);
}
```

ReducerIR

```
entry:
  %x = alloca reducer()
  br for

for:
  %idx = phi [0, entry], [%idx.next, cont]
  %idx.next = %idx + 1
  br %idx == 10, body, exit

body:
  detach det, cont

det:
  %prev = load %x
  store %x = %prev + 1
  reattach cont

cont:
  br for

exit:
  sync
  %final = load %x
  return %final
```

ReducerIR - Primitive Proof of Correctness

- Detach and sync has union of read/write semantics of body
  - Only loads of reducer variables that can be hoisted up are those not written to inside loop => Legal
  - Only writes that can be hoisted up are those not loaded within loop => Legal
  - Talk offline for more proof details
- Existing serial optimizations work without issue from memory semantics^
- Any parallelization-modifying or serialization passes require special care
  - May need to use the reducer function attribute

ReducerIR - Optimization Benefits

```
cilk_for(int i=0; i<n; i++)
  // separate lookup every iteration
  REDUCER_VIEW(total)++;

return REDUCER_VIEW(total);
```

```
cilk_for(int i=0; i<n; i+=10)
  for(int j=0; i<min(i+10,n); j++)
    // 10 x extraneous lookups
    REDUCER_VIEW(total)++;

return REDUCER_VIEW(total);
```

```
// further becomes += n with primitive serialization pass
cilk_for(int i=0; i<n; i+=10)
   REDUCER_VIEW(total)+=min(i+10,n);

return n;
```

ReducerIR - Optimization Benefits

- Add example:
  - Reducer without optimization: 0.24s
  - Reducer with optimization: <0.01s

Conclusions

- Better semantics are needed for reducers at both linguistic and compiler level
- Provide benefits in both understanding and optimizing code
- In both cases, reasonable representation derived from modification to memory behavior
  - LValue references in front-end
  - Allocation in middle-end
- Prototype Implementation in Tapir @ LLVM8

ReducerIR - Optimization Benefits (Slide incomplete)

```
reducer<std::vector<int>> total = {0, 0};

cilk_for(int i=0; i<n; i++) {
  auto& view = REDUCER_VIEW(total);
  for(int j=0; j<total(); j++) {
     view[j]++;
  }
}
return REDUCER_VIEW(total);
```

```
size = total.size(); // move out of loop since size doesn't change
cilk_for(int i=0; i<n; i++) {
  auto& view = REDUCER_VIEW(total);
  for(int j=0; j<size; j++) {
     view[j]++;
  }
}
return REDUCER_VIEW(total);
```