EASY: Efficient Arbiter SYnthesis from Multi-threaded Code

Authors: Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason H. Anderson and George A. Constantinides

Presenter: Jianyi Cheng
Multi-Threaded Code using PThreads

```c
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```

Two `accum` threads running in parallel:

- Thread 0  touches B[0: 511]
- Thread 1  touches B[512: 1023]
Multi-Threaded Code using PThreads

```c
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```

// Thread 0
```c
void accum( 0 ) {
    for i = 0 to 511
        temp += B[i];
}
```

// Thread 1
```c
void accum( 512 ) {
    for i = 512 to 1023
        temp += B[i];
}
```

Two `accum` threads running in parallel:

- Thread 0 touches B[0: 511]
- Thread 1 touches B[512: 1023]
Motivation

Low memory bandwidth

=> memory contention

![Diagram](image-url)

Input array [4096]

T0 T1 T2 T3 T4 T5 T6 T7

○: arbiter

T: thread
Prior Work: Profiling-Based Array Partitioning

At runtime simulation…

Input array [4096]


Yanyi Cheng17@imperial.ac.uk
Prior Work: Profiling-Based Array Partitioning

Hopefully they will behave like this…

Partitioned input array [512] x 8

Prior Work: Profiling-Based Array Partitioning

But not guaranteed, so still need arbitration.

Partitioned input array [512] x 8

Prior Work: Profiling-Based Array Partitioning

What a mess!

Partitioned input array [512] x 8

P0  P1  P2  P3  P4  P5  P6  P7

T0  T1  T2  T3  T4  T5  T6  T7

○: arbiter  T: thread  P: partitioned memory

Motivation
It can get worse…

Area and fmax of total arbitration logic

- LUT count
- Reg count
- Fmax

Number of threads and partitions
Hardware resources

Maximum clock frequency - MHz
Motivation

It can get worse...

Area and fmax of total arbitration logic

Arbitration hurts area and kills max clock frequency!
Motivation

What a mess!

Partitioned input array [512] x 8

○: arbiter   T: thread   P: partitioned memory


jianyi.cheng17@imperial.ac.uk
Motivation

If we can prove...

Partitioned input array [512] x 8

P0  P1  P2  P3  P4  P5  P6  P7

T0  T1  T2  T3  T4  T5  T6  T7

\(\bigcirc\): arbiter  \(T\): thread  \(P\): partitioned memory
Motivation

Then we have...

Partitioned input array [512] x 8

- P0, P1, P2, P3, P4, P5, P6, P7
- T0, T1, T2, T3, T4, T5, T6, T7

○: arbiter
T: thread
P: partitioned memory
Motivation

Or even **simple** like this…

Partitioned input array [512] x 8

- **P0**, **P1**, **P2**, **P3**, **P4**, **P5**, **P6**, **P7**
- **T0**, **T1**, **T2**, **T3**, **T4**, **T5**, **T6**, **T7**

○: arbiter    T: thread    P: partitioned memory
Efficient Arbiter SYNthesis

An EASY way to take arbiters away
Research Contributions

- Formal methods to prove memory bank exclusivity
- Automated removal or radical simplification of arbitration
- Up to 87% area saving
- Up to 39% wall-clock time improvement
Implementation

Clang

C code

LLVM IR code

IR transformations

Pass 1
Pass 2
... Array partitioning pass

Optimized LLVM IR code 0

Optimized LLVM IR code 1

RTL code

LegUp backend

Arbiter optimization

Extracted partitioned memory behaviour

Verification code

EASY: Efficient Arbiter Synthesis from Multi-threaded Code
jianyi.cheng17@imperial.ac.uk
Microsoft Boogie

Intended to formally verify a single-threaded program

Built on top of SMT solvers

Uses its own intermediate verification language (IVL)

Automatically verified by Boogie ‘behind the scenes’, hidden from the user

This work: We show that it can be used for arbitration simplification of multi-threaded code
Microsoft Boogie

- **assert c**
  instructs the verifier to try to prove the condition c.

- **havoc x**
  assigns an arbitrary value to the variable x.

- **assume c**
  tells the verifier that condition c can be assumed true.

- **if(*) {A} else {B}**
  tells the verifier that either branch might be taken arbitrarily.
Multi-Threaded Code using PThreads

void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}

// Thread 0
void accum(0) {
    for i = 0 to 511
        temp += B[i];
}

// Thread 1
void accum(512) {
    for i = 512 to 1023
        temp += B[i];
}

Two accum threads running in parallel:

- Thread 0 touches B[0: 511]
- Thread 1 touches B[512: 1023]
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}

Two **accum** threads running in parallel:

- Thread 0 touches B[0: 511]
- Thread 1 touches B[512: 1023]

**Block partitioning scheme:**

For any $i$ in $[0:511]$:

- $i >> 9$ (MSB) is always 0

For any $i$ in $[512: 1023]$:

- $i >> 9$ (MSB) is always 1

**Partition index = $i >> 9$**
Methodology

Input Code

```c
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```

- Thread 0 touches B[0:511]
- Thread 1 touches B[512:1023]
Methodology

Program slicing - All we need is memory behavior

LLVM IR Code

Input Code

```c
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```

```c
void accum(int N) {
    for i = N to N+511
        load B[i];
        load temp;
        temp_new = temp + B[i];
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```
Methodology

Loop invariants

Input Code

```c
void accum(int N) {
    for i = N to N+511 {
        load B[i];
    }
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```

Code with invariants

```c
void accum(int N) {
    for i = N to N+511 {
        assert N <= i <= N+511;
        temp += B[i];
        assert N <= i <= N+511;
    }
}

int main(void) {
    run `accum` with N = 0;
    run `accum` with N = 512;
}
```
Methodology

Input Code

```
void accum(int N) {
    for i = N to N+511
        load B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}
```

Boogie Code

```
procedure accum(N) returns (read, partition_index) {
    assert i >= N && i <= N+511;
    havoc i;
    assume i >= N && i <= N+511;
    partition_index = i >> 9;
    if(*){
        read = true;
        return;
    }
    assert i >= N && i <= N+511;
    read = false;
    return;
}
```
Methodology

Input Code

```c
void accum(int N) {
    for i = N to N+511
        load B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}
```

Boogie Code

```boogie
procedure accum(N) returns (read, partition_index) {
    assert i >= N && i <= N+511;
    havoc i;
    assume i >= N && i <= N+511;
    partition_index = i >> 9;
    if(*){
        read = true;
        return;
    }
    assert i >= N && i <= N+511;
    read = false;
    return;
}
```
Methodology

Input Code

```c
void accum(int N) {
    for i = N to N+511
        load B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}
```

Boogie Code

```boogie
procedure accum(N) returns (read, partition_index) {
    assert i >= N && i <= N+511;
    havoc i;
    assume i >= N && i <= N+511;
    partition_index = i >> 9;
    if(*){
        read = true;
        return;
    }
    assert i >= N && i <= N+511;
    read = false;
    return;
}
```
Methodology

Input Code

```c
void accum(int N) {
    for i = N to N+511
        load B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}
```

Boogie Code

```c
procedure main() {
    call t0_read, t0_index = assign(0);
    call t1_read, t1_index = assign(512);

    // T - thread; B - memory bank
    // To verify T0 never access B0 - ×
    assert !t0_read || t0_index != 0;
    // To verify T0 never access B1 - ✓
    assert !t0_read || t0_index != 1;

    // To verify T1 never access B0 - ✓
    assert !t1_read || t1_index != 0;
    // To verify T1 never access B1 - ×
    assert !t1_read || t1_index != 1;
}
```
Methodology

Input Code

```c
void accum(int N) {
    for i = N to N+511
        load B[i];
}

int main(void) {
    run accum with N = 0;
    run accum with N = 512;
}
```

Boogie Code

```python
procedure main() {
    call t0_read, t0_index = assign(0);
    call t1_read, t1_index = assign(512);

    // T - thread; B - memory bank
    // To verify T0 never access B0 - ×
    assert !t0_read || t0_index != 0;
    // To verify T0 never access B1 - ✓
    assert !t0_read || t0_index != 1;

    // To verify T1 never access B0 - ✓
    assert !t1_read || t1_index != 0;
    // To verify T1 never access B1 - ×
    assert !t1_read || t1_index != 1;
}
```

EASY: Efficient Arbiter Synthesis from Multi-threaded Code

jianyi.cheng17@imperial.ac.uk
Methodology

- Take whole program & automatically transform into Boogie
- Support any form of memory access patterns
- Each thread can return partition index of any iteration
- Verify assertions with all possible memory accesses
- Remove arbiters or simplify with fewer ports
Case study: Histogram Benchmark

- # of banks = # of threads
- Non-overlapping accesses
  => All arbiters removable
- Analyze range of array data
- Construct histogram

PB: partitioned bank  AB: arbiter
THD: thread

PB0  PB1  PB2  PB3

THD0  THD1  THD2  THD3

jianyi.cheng17@imperial.ac.uk
Case study: Histogram Benchmark

- LUT before
- LUT after
- Wall clk time before
- Wall clk time after

10442 (58% saving)
51ms (24% accel.)

EASY: Efficient Arbiter Synthesis from Multi-threaded Code
jianyi.cheng17@imperial.ac.uk
Results

Improvements on the total hardware

<table>
<thead>
<tr>
<th>Function</th>
<th>LUT</th>
<th>Fmax</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>58%</td>
<td>32%</td>
<td>4%</td>
</tr>
<tr>
<td>matrixadd</td>
<td>84%</td>
<td>45%</td>
<td>0%</td>
</tr>
<tr>
<td>matrixmult</td>
<td>83%</td>
<td>16%</td>
<td>0%</td>
</tr>
<tr>
<td>matrixmult(cyclic)</td>
<td>46%</td>
<td>12%</td>
<td>0%</td>
</tr>
<tr>
<td>matrixtrans</td>
<td>87%</td>
<td>50%</td>
<td>3%</td>
</tr>
<tr>
<td>matrixtrans(blockcyclic)</td>
<td>14%</td>
<td>0%</td>
<td>2%</td>
</tr>
<tr>
<td>substring</td>
<td>77%</td>
<td>62%</td>
<td>1%</td>
</tr>
<tr>
<td>los</td>
<td>13%</td>
<td>10%</td>
<td>2%</td>
</tr>
</tbody>
</table>

geo. mean 58% 28% 2%
Conclusion

Summary

• Multi-threaded code => Single-threaded Boogie
• Arbitrary input code support
• Automation of arbiter verification and simplification
• Verification time - 13s av. & 70s max
• It may not improve the design but never get worse

Future work

• Complex while loops
• Memory interaction between threads
• Indirect array indexing
Efficient Arbiter SYnthesis

An EASY way to take arbiters away