Imperial College London

25-02-2019

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

```
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;
}
```

in parallel

Two ***accum*** threads running in parallel:

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

jianyi.cheng17@imperial.ac.uk

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;
}
```

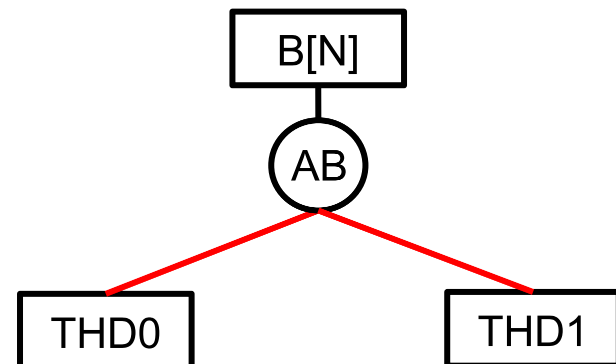
in parallel

```
// 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]

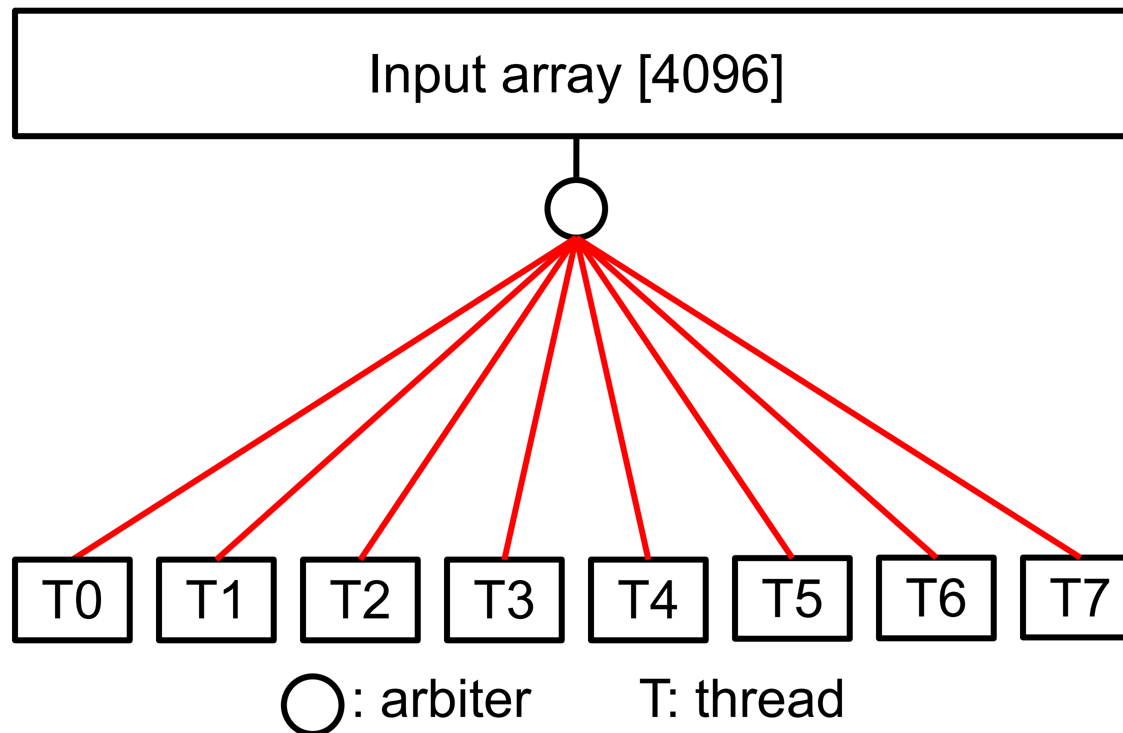


jianyi.cheng17@imperial.ac.uk

Motivation

Low **memory bandwidth**

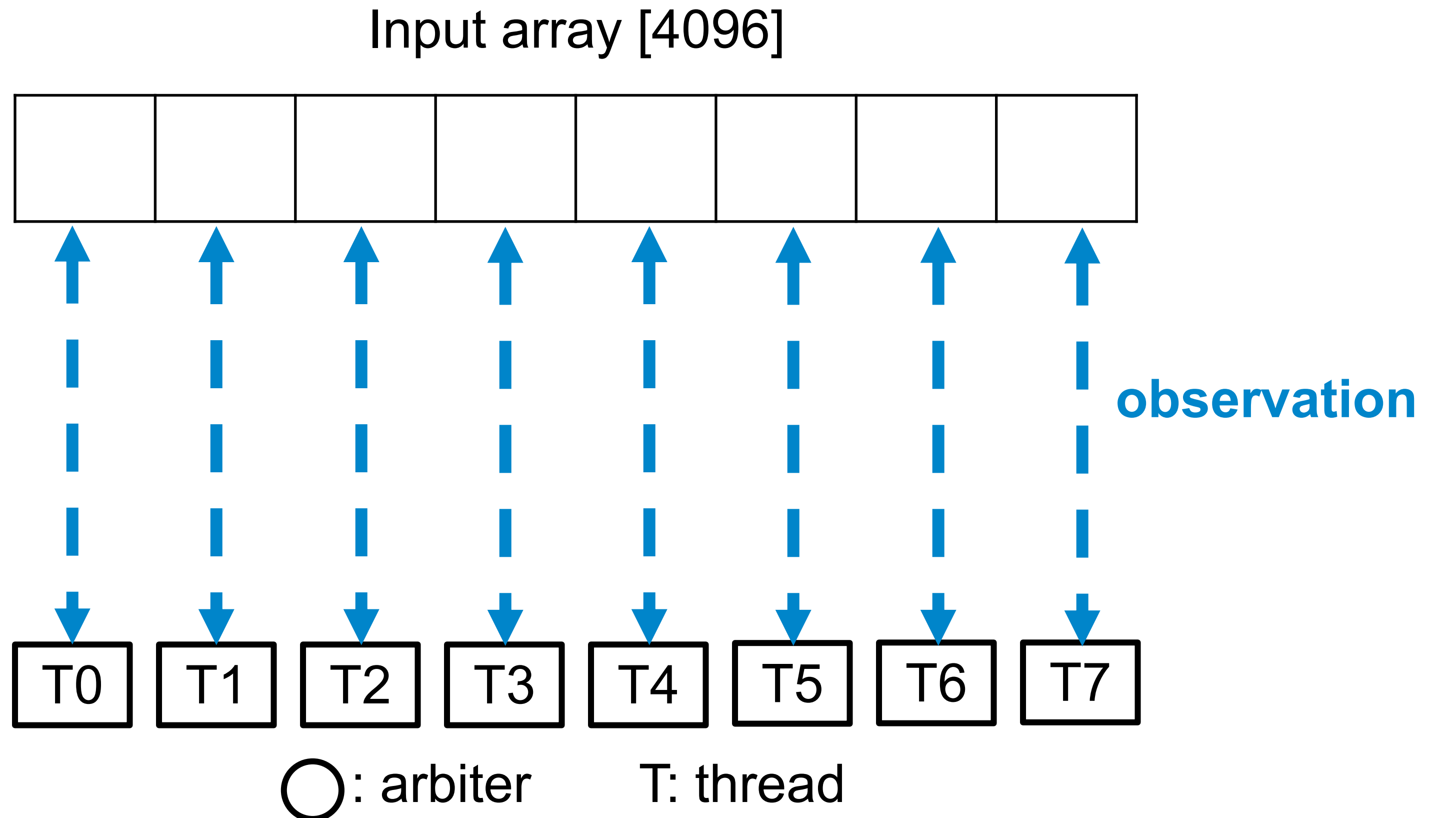
=> memory contention



jianyi.cheng17@imperial.ac.uk

Prior Work: Profiling-Based Array Partitioning

At **runtime simulation**…

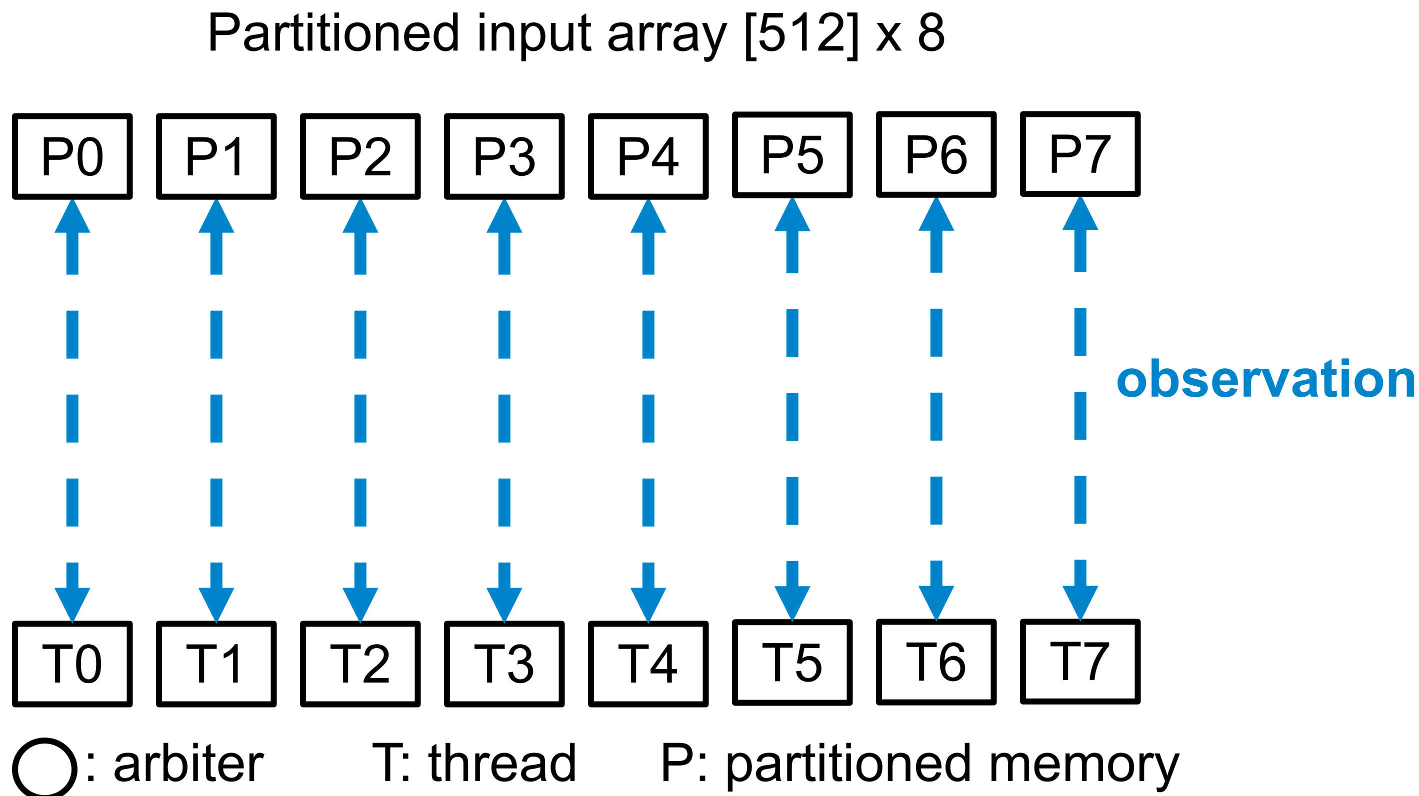


Y-T. Chen and J. H. Anderson, "Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software," *FPL*, Ghent, Belgium, 2017.

 jianyi.cheng17@imperial.ac.uk

Prior Work: Profiling-Based Array Partitioning

Hopefully they will behave like this…

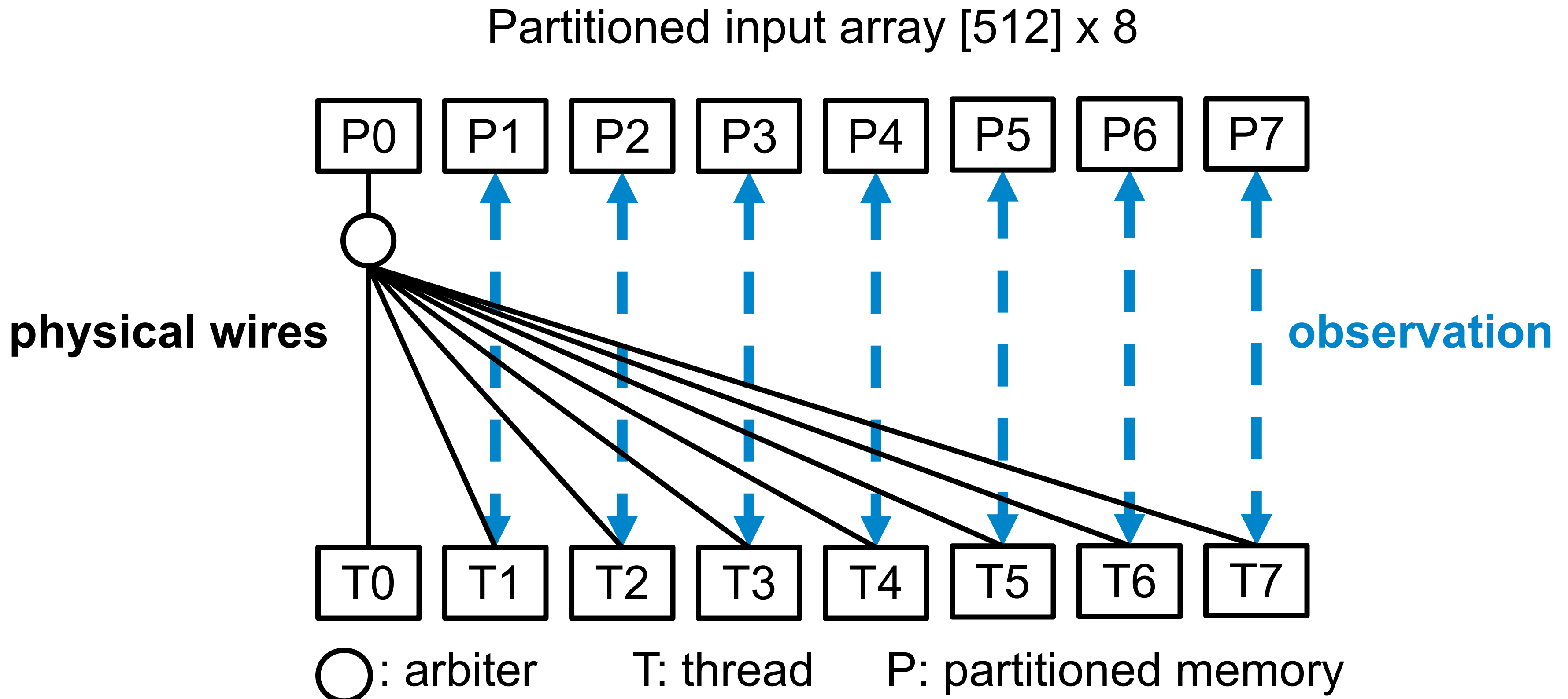


Y-T. Chen and J. H. Anderson, "Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software," *FPL*, Ghent, Belgium, 2017.

jianyi.cheng17@imperial.ac.uk

Prior Work: Profiling-Based Array Partitioning

But **not guaranteed**, so still need arbitration.

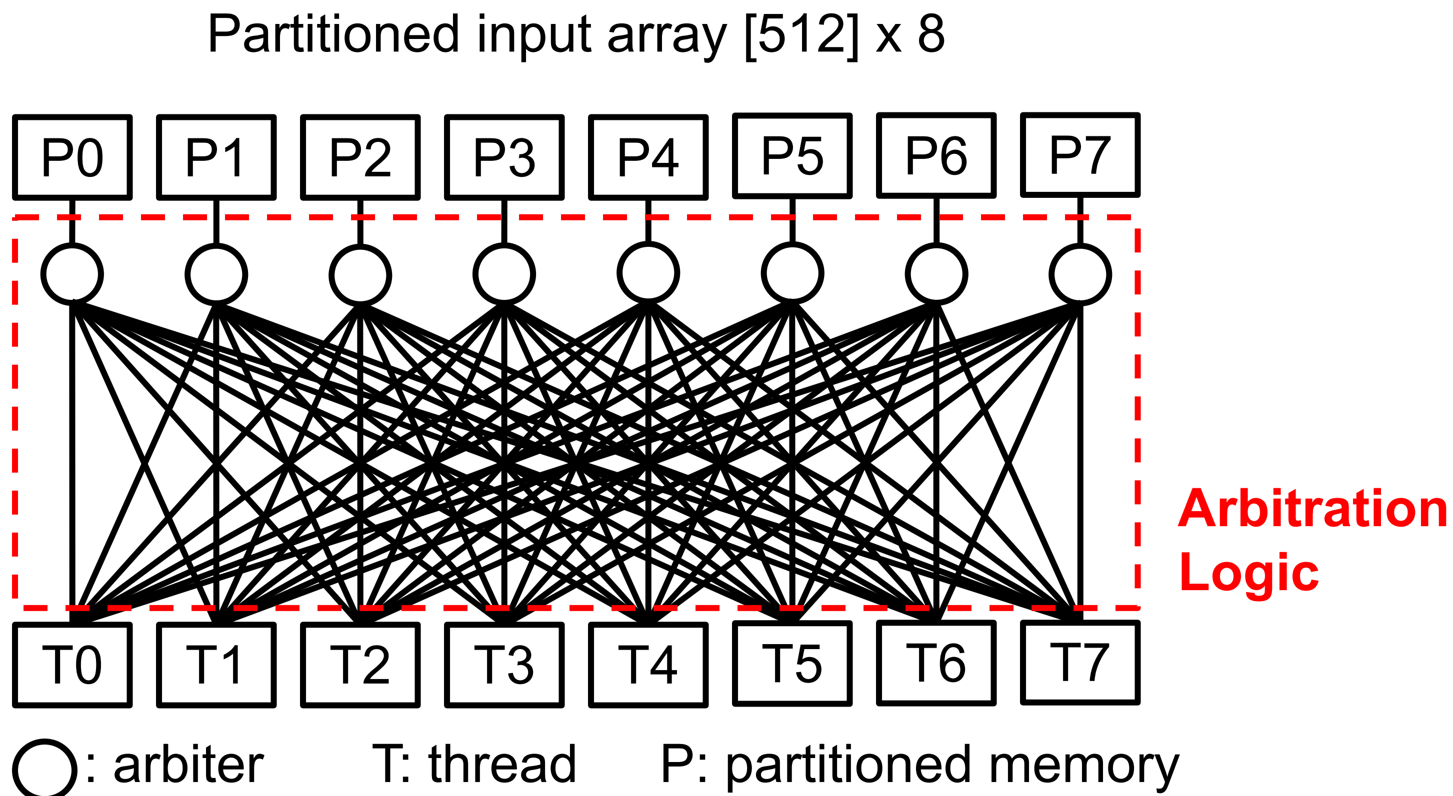


Y-T. Chen and J. H. Anderson, "Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software," *FPL*, Ghent, Belgium, 2017.

 jianyi.cheng17@imperial.ac.uk

Prior Work: Profiling-Based Array Partitioning

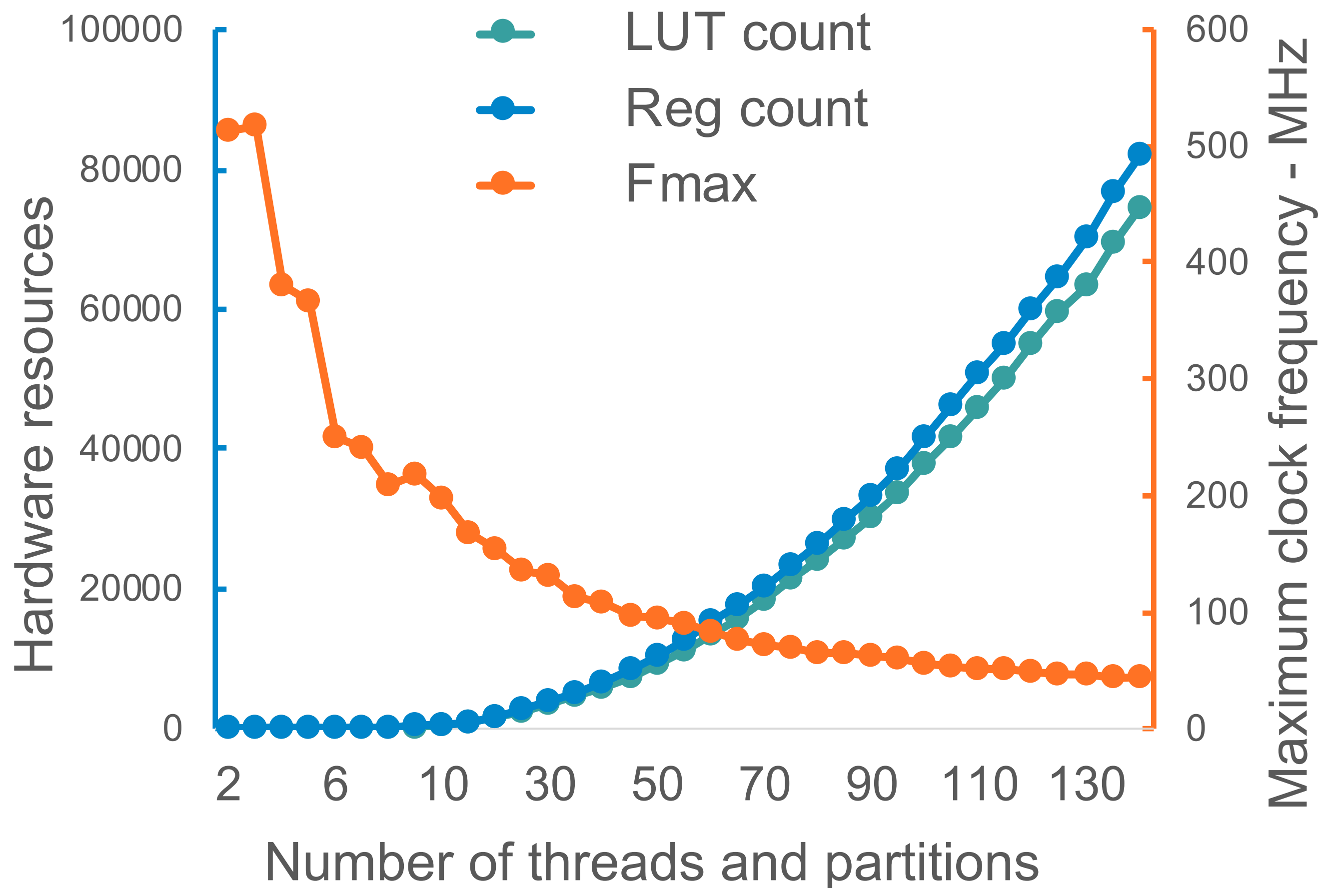
What a **mess!**



Y-T. Chen and J. H. Anderson, "Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software," *FPL*, Ghent, Belgium, 2017.

Motivation

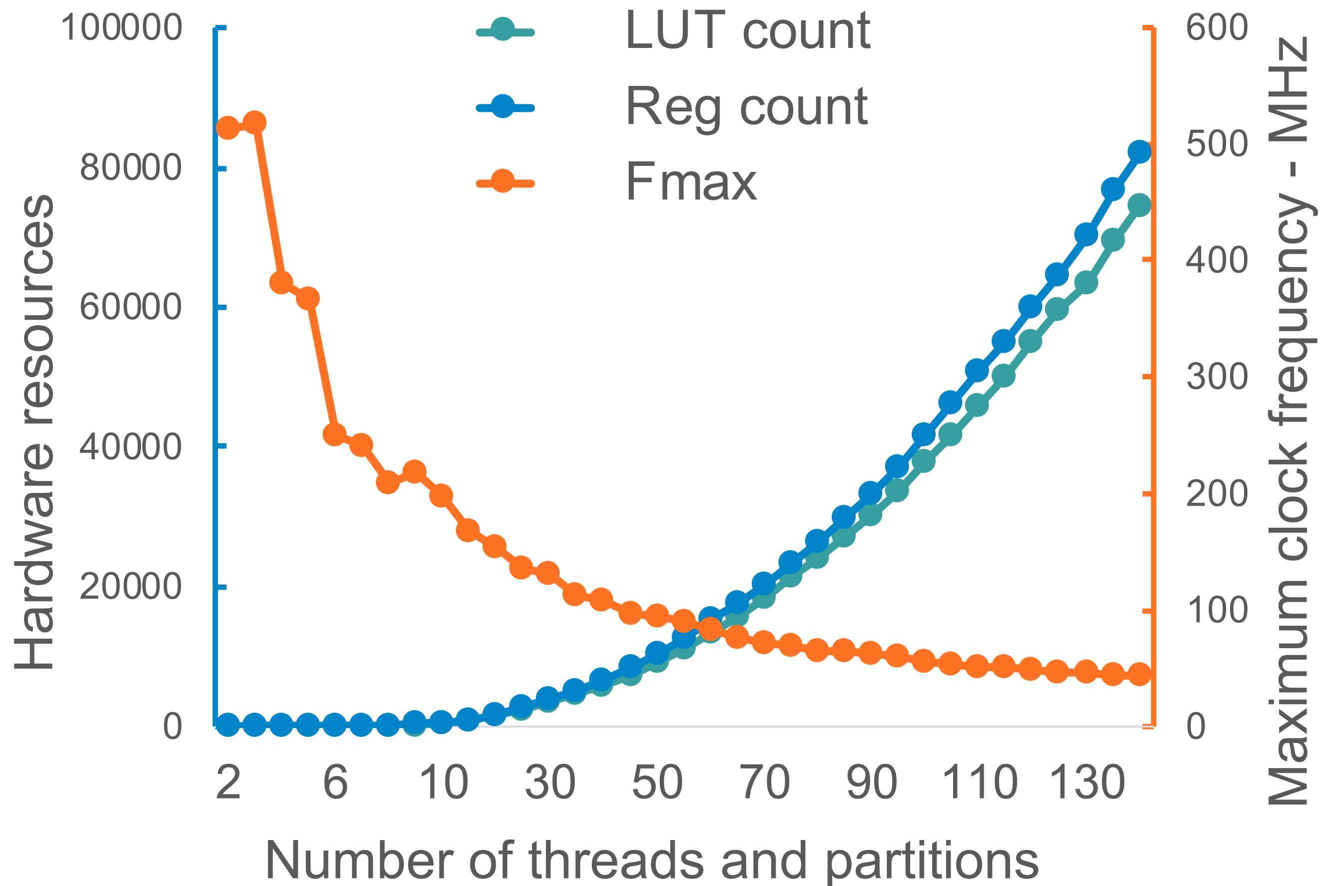
It can get **worse**…



jianyi.cheng17@imperial.ac.uk

Motivation

It can get **worse**…

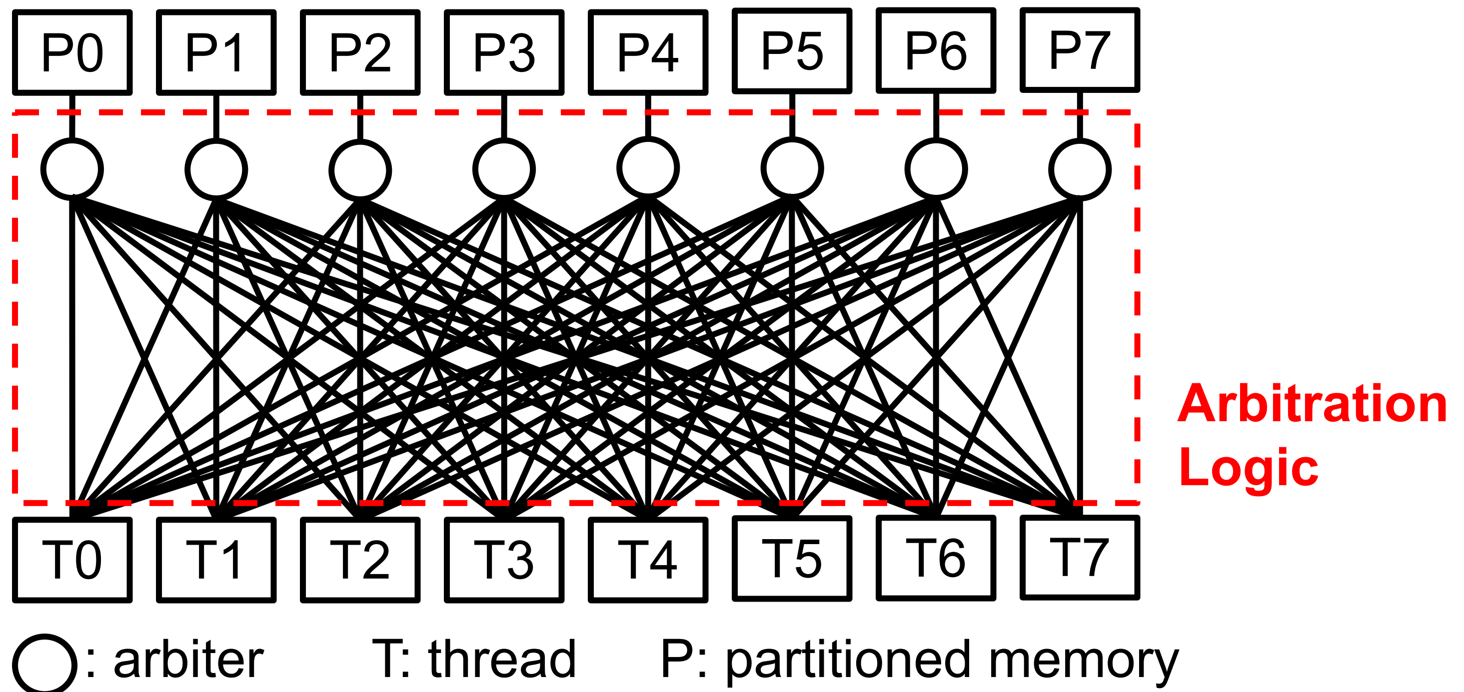


Arbitration hurts area and kills max clock frequency!

jianyi.cheng17@imperial.ac.uk

Motivation

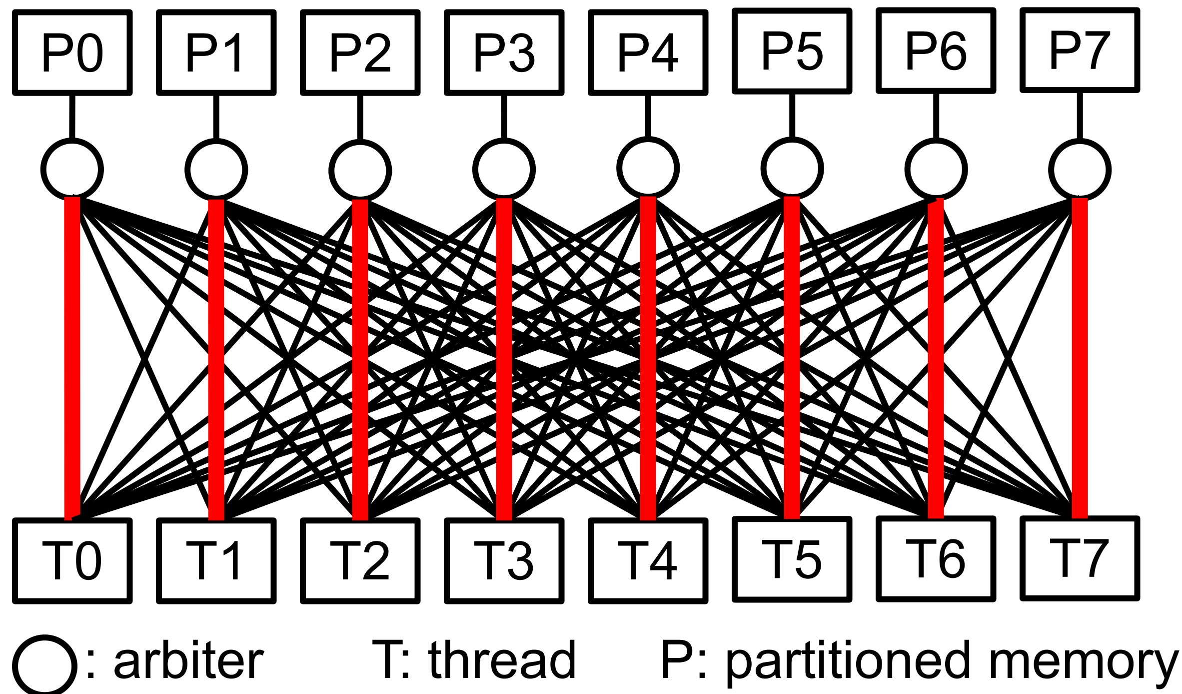
What a **mess!**



Y-T. Chen and J. H. Anderson, "Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software," *FPL*, Ghent, Belgium, 2017.

Motivation

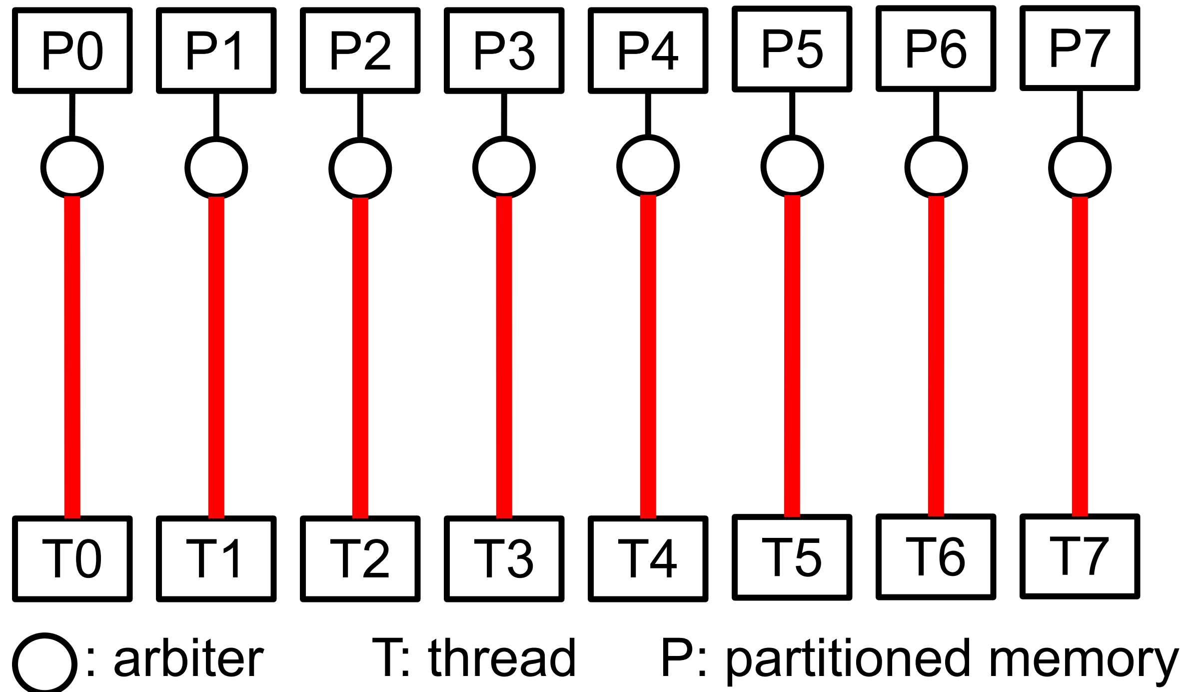
If we can **prove**…



jianyi.cheng17@imperial.ac.uk

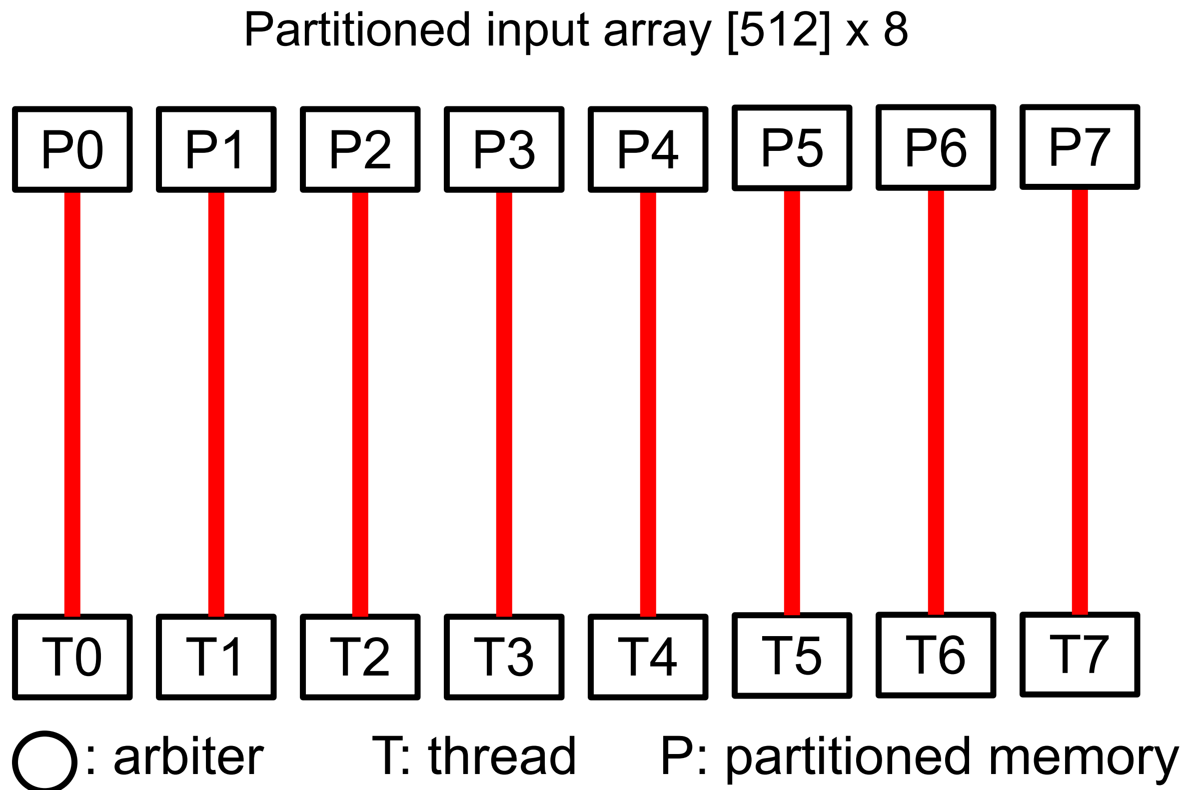
Motivation

Then we have…



Motivation

Or even **simple** like this…



jianyi.cheng17@imperial.ac.uk

Efficient Arbiter SYnthesis

An EASY way to take arbiters away

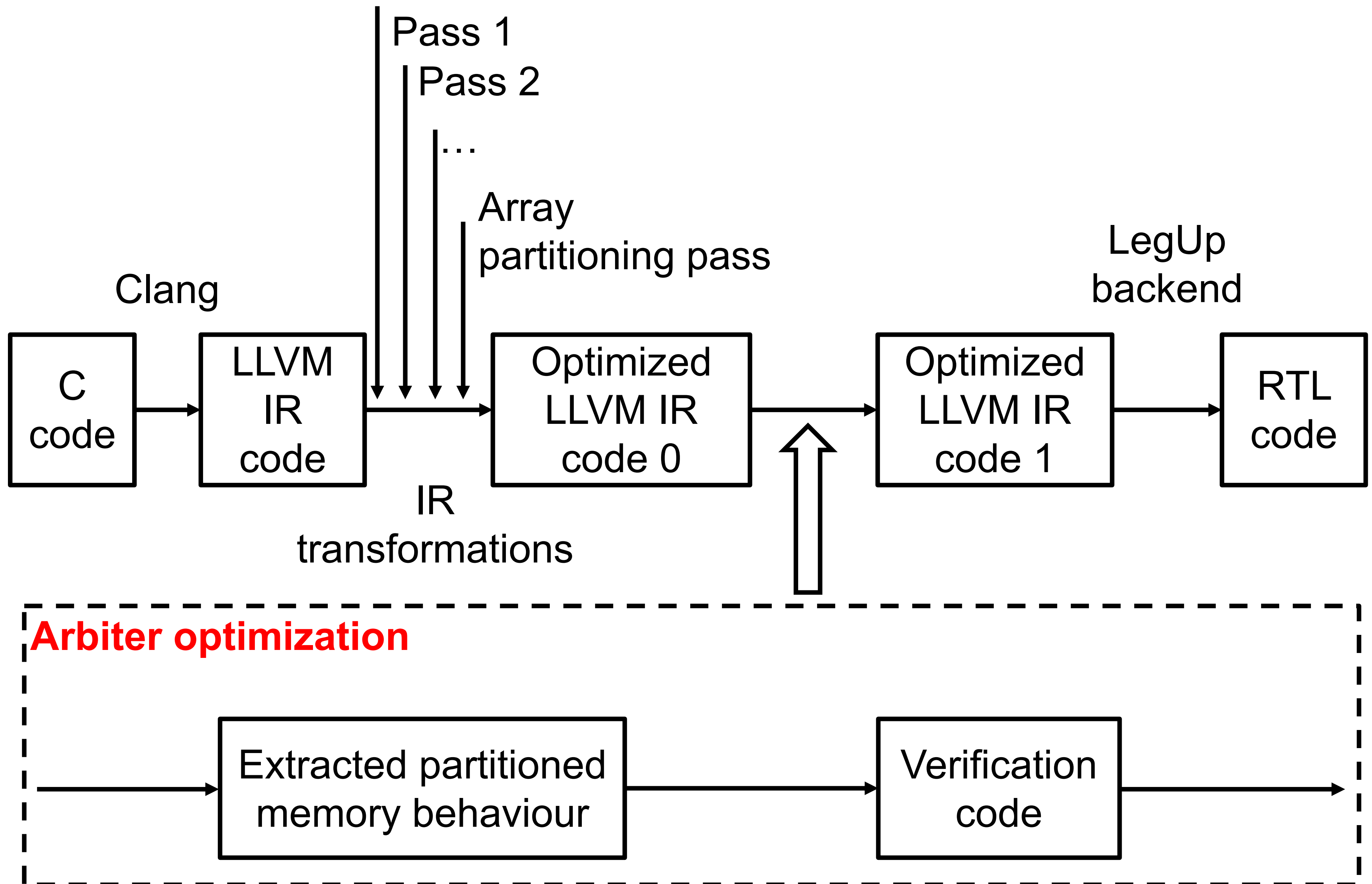
jianyi.cheng17@imperial.ac.uk

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

jianyi.cheng17@imperial.ac.uk

Implementation



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

jianyi.cheng17@imperial.ac.uk

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.

jianyi.cheng17@imperial.ac.uk

Multi-Threaded Code using PThreads

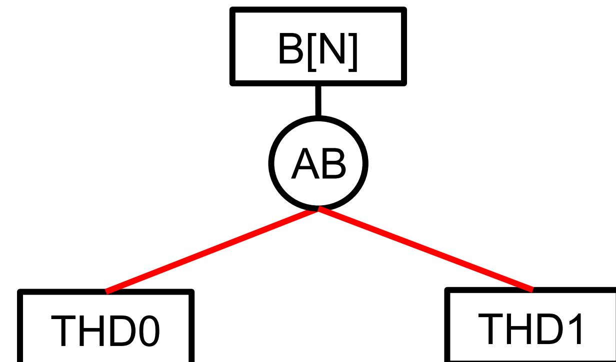
```
void accum(int N) {
    for i = N to N+511
        temp += B[i];
}

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]



jianyi.cheng17@imperial.ac.uk

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;
}
```

Block partitioning scheme:

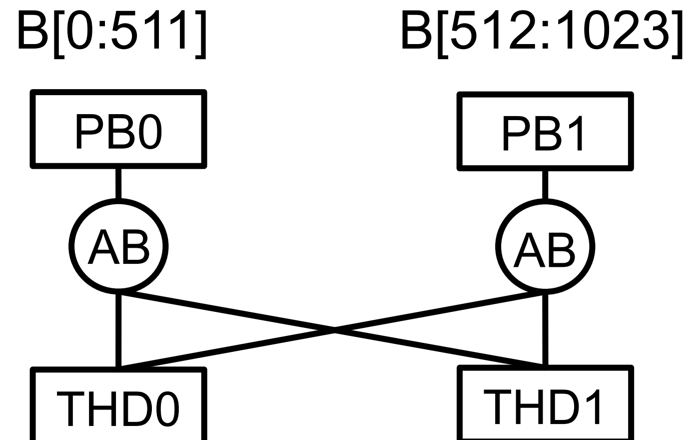
For any **i** in [0:511]:
i >> 9 (MSB) is always 0

For any **i** in [512: 1023]:
i >> 9 (MSB) is always 1

Partition index = i >> 9

Two ***accum*** threads running in parallel:

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



jianyi.cheng17@imperial.ac.uk

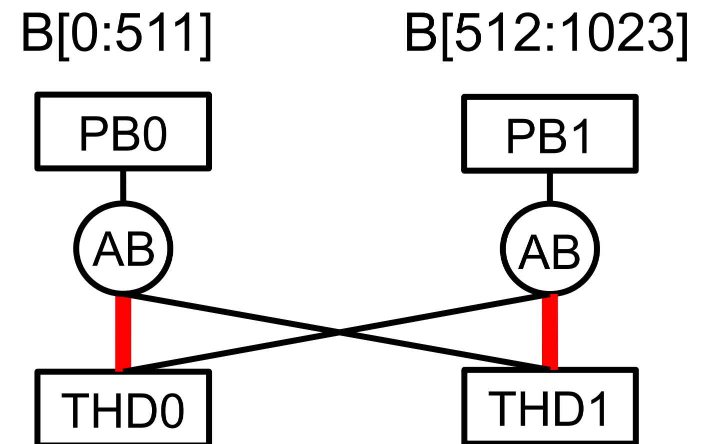
Methodology

Input Code

```
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]

Resultant hardware



jianyi.cheng17@imperial.ac.uk

Methodology

Program slicing - All we need is memory behavior

Input Code

```
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;
}
```

LLVM IR Code

```
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;
}
```

ignored

jianyi.cheng17@imperial.ac.uk

Methodology

Loop invariants

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;
}
```

Code with invariants

```
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;
}
```

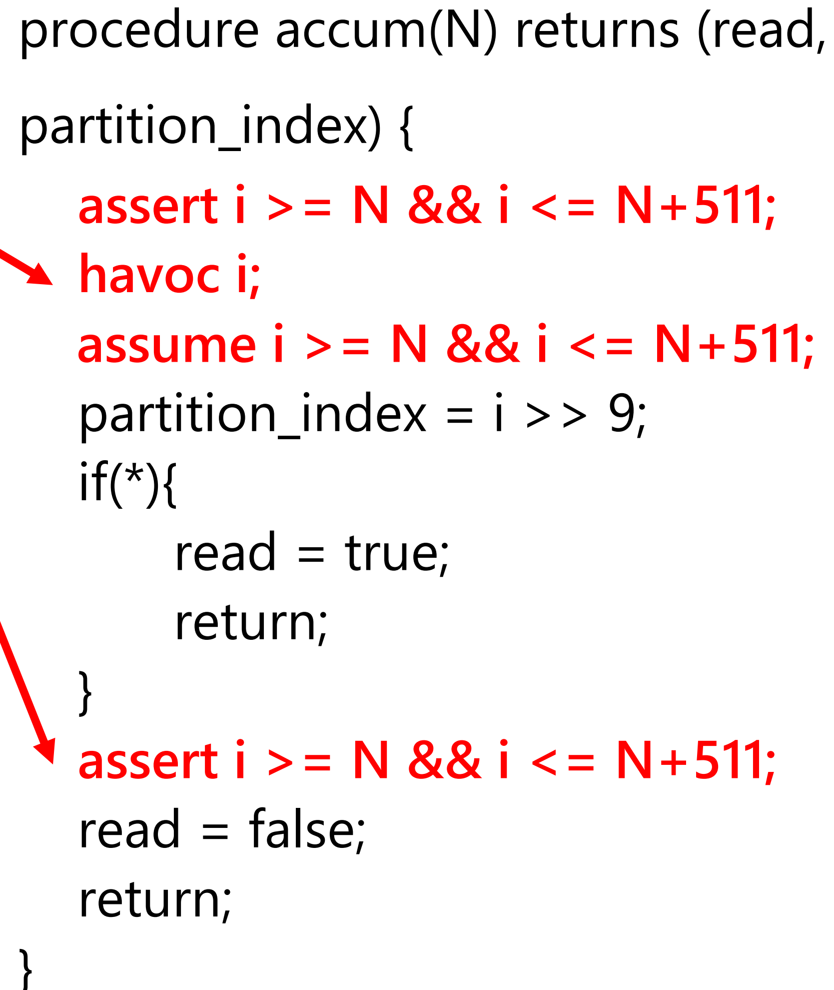

jianyi.cheng17@imperial.ac.uk

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

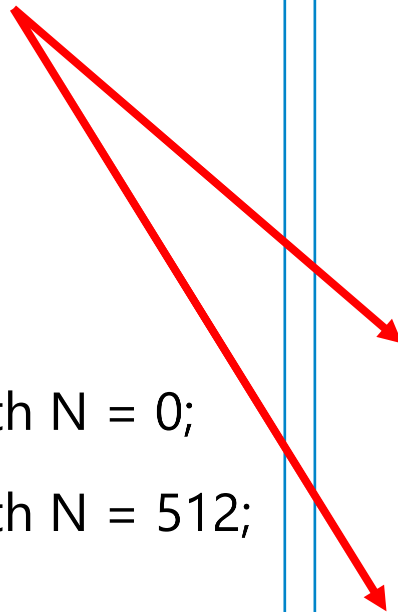


jianyi.cheng17@imperial.ac.uk

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

```
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;
}
```


jianyi.cheng17@imperial.ac.uk

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 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;
}
```


jianyi.cheng17@imperial.ac.uk

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 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;
}
```

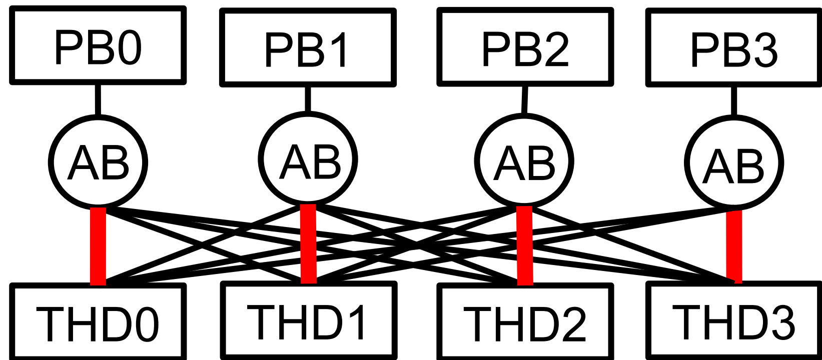

jianyi.cheng17@imperial.ac.uk

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

jianyi.cheng17@imperial.ac.uk

Case study: Histogram Benchmark



PB: partitioned bank AB: arbiter
THD: thread

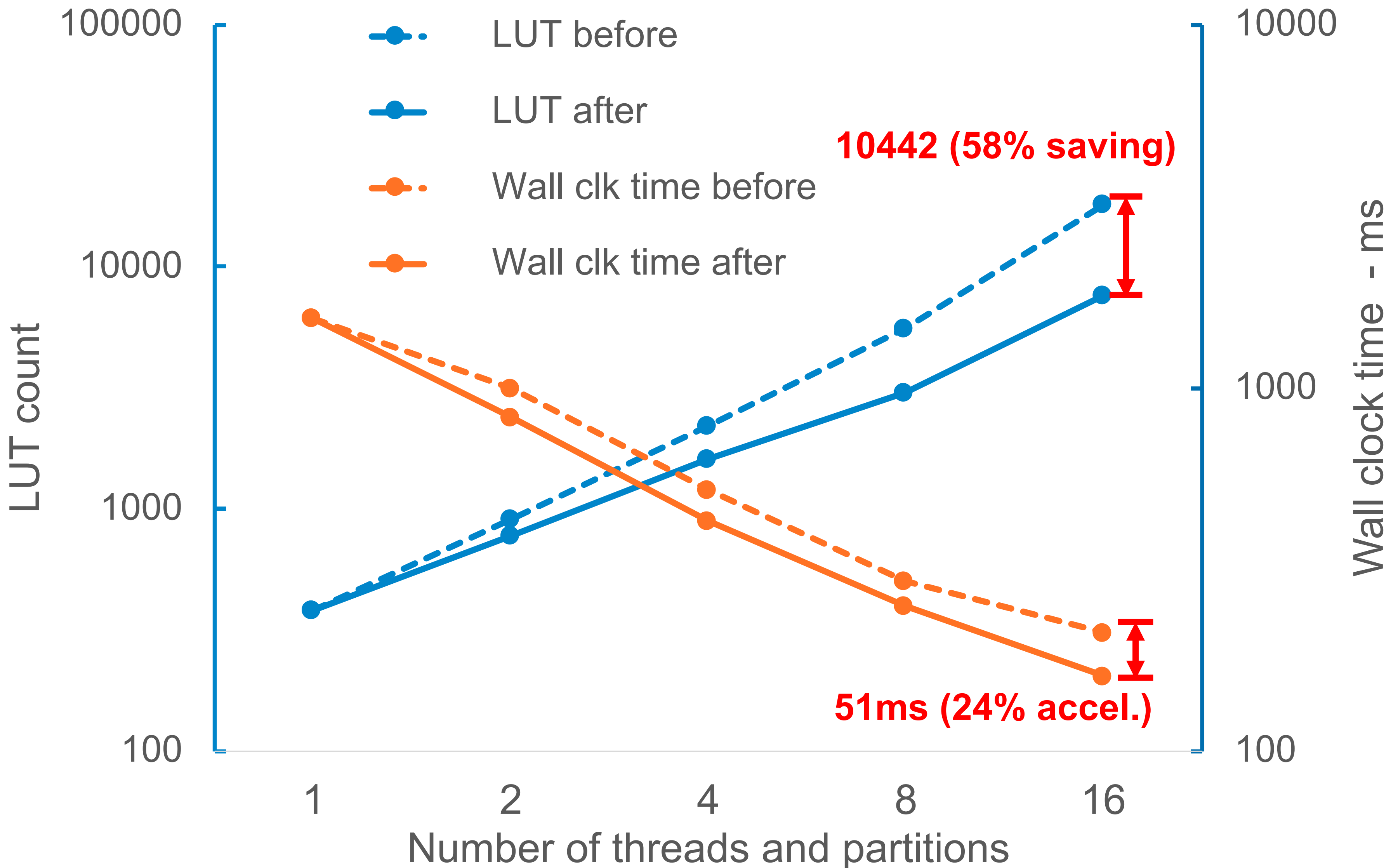
- # of banks = # of threads
- Non-overlapping accesses

  => All arbiters removable

- Analyze range of array data
- Construct histogram

jianyi.cheng17@imperial.ac.uk

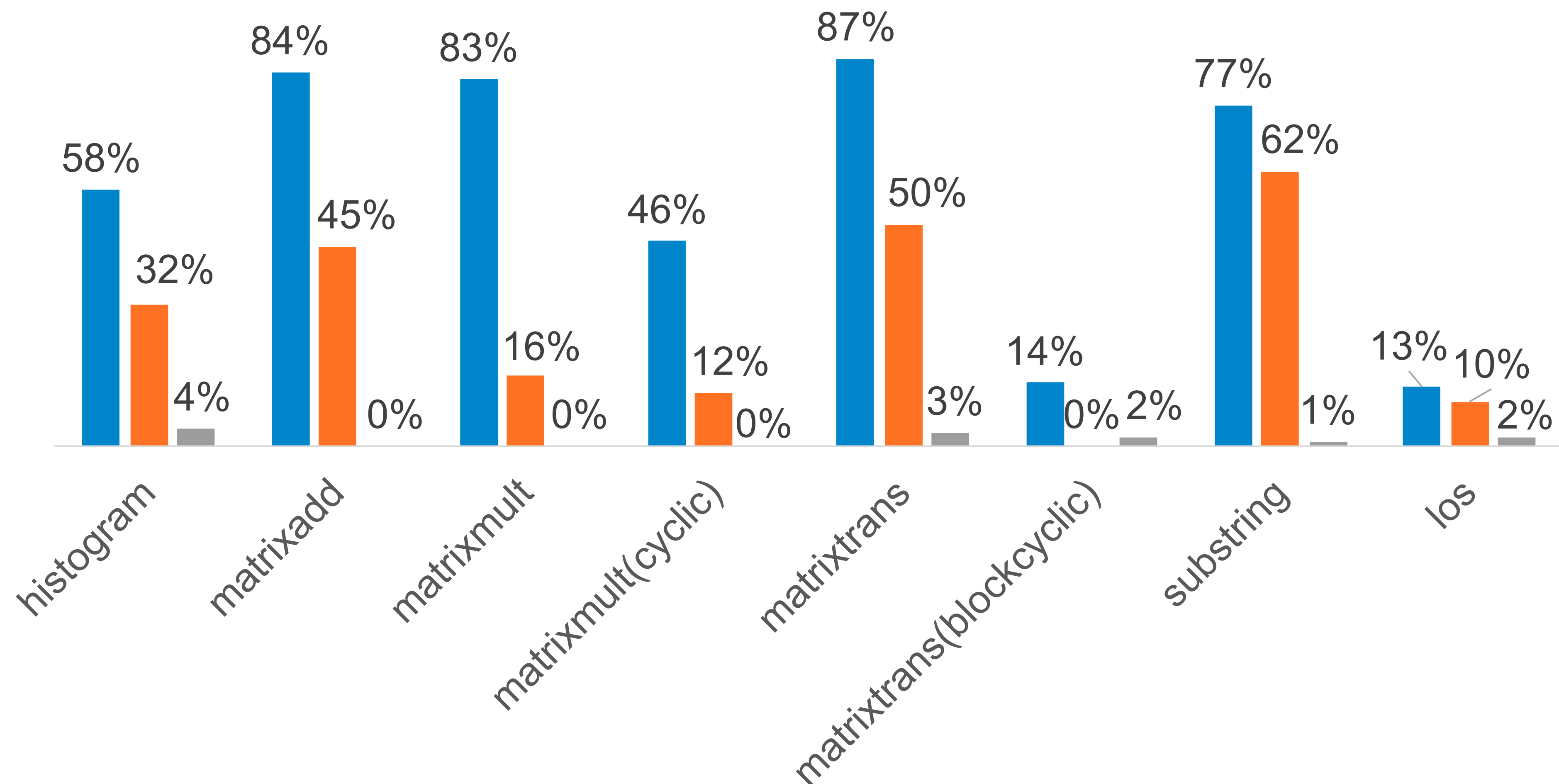
Case study: Histogram Benchmark



jianyi.cheng17@imperial.ac.uk

Results

Improvements on the total hardware



jianyi.cheng17@imperial.ac.uk

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

jianyi.cheng17@imperial.ac.uk

Efficient Arbiter SYnthesis

An EASY way to take arbiters away