Loop Fusion in LLVM

Under the guidance of:
Prof. H D Nandeesh

Presented by-
Madhura Dinesh Kaushik
Sridhar G
Supreeth M S

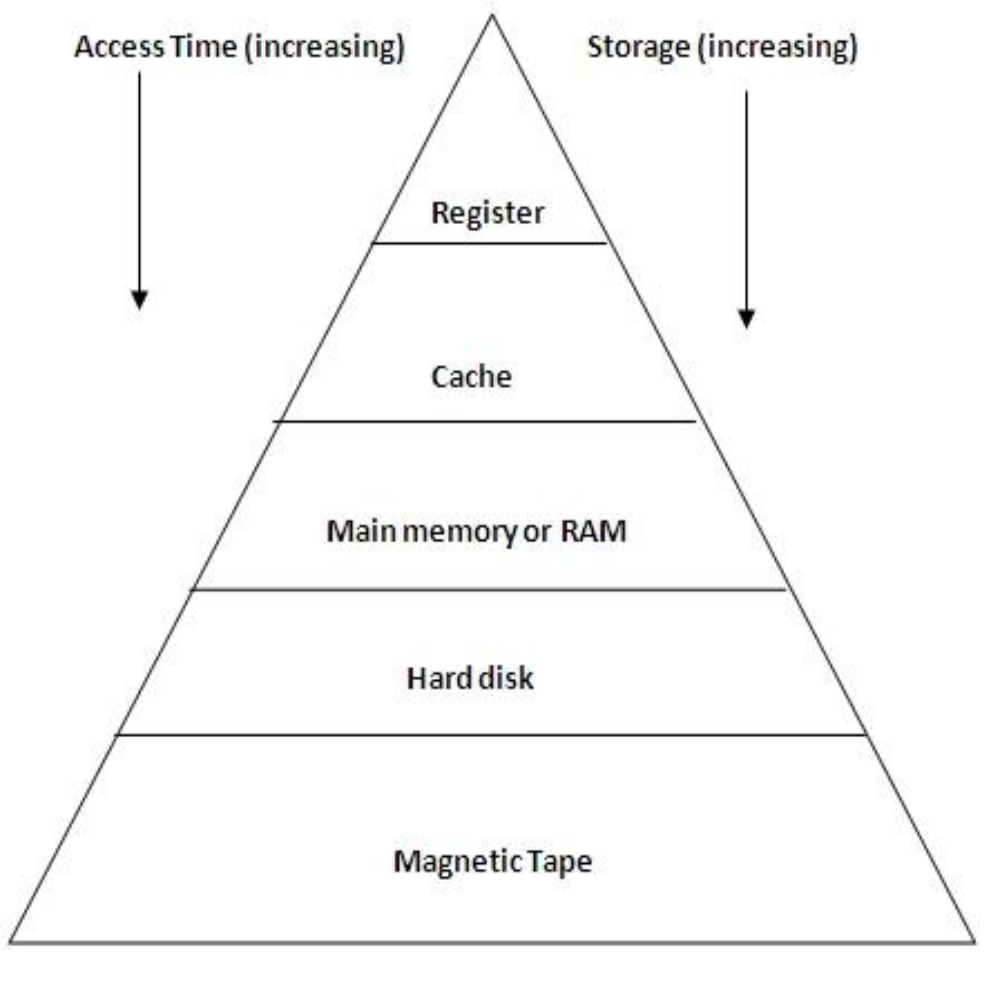
Motivating Example

```
for( i = 0; i < 1000; i++ )
{
    sum += a[i];
}
for( j = 0; j < 1000; j++ )
{
    prod *= a[j];
}
```

```
for( i = 0; i < 1000; i++ )
{
    sum += a[i];
    prod *= a[i];
}
```

What is Cache?

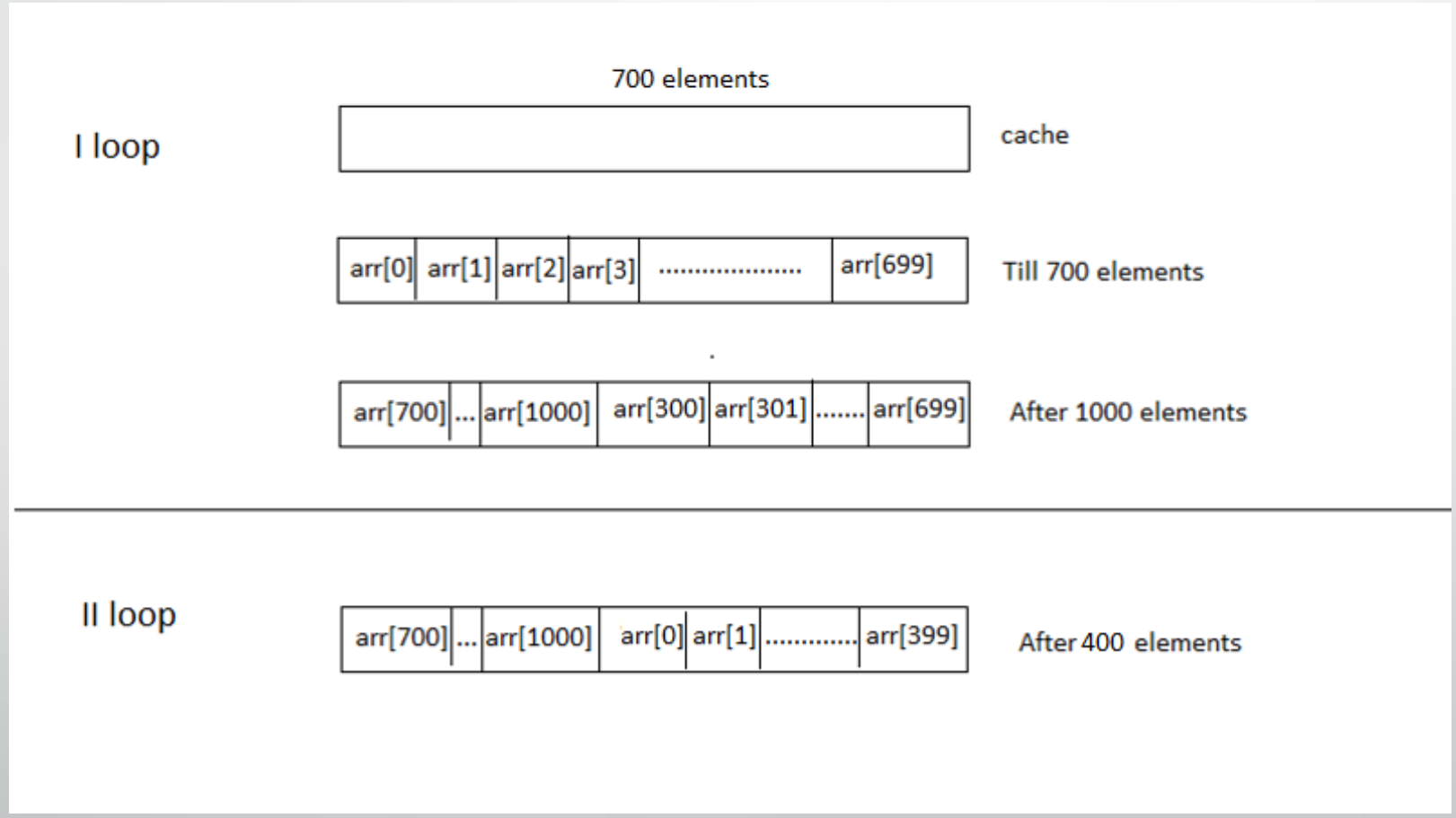
- Cache memory is a fast accessible memory space
- Lies between CPU and Main Memory
- Stores recently accessed data for future requests



Issues with Cache

- Cache space is limited
- Elements in cache need to be replaced when new data is accessed
- In our example, what if array size is greater than the cache size?

Issues with Cache



Concept of Loop Fusion

- Combining the body of two loops into a single loop
- This will:
  - Reduce the number of memory accesses
  - Reduce run time
  - Increase memory reuse

Is it Legal?

- Valid only if:
  - loops are adjacent
  - Index variables runs within the same bound
  - Result does not change after fusing
  - NO anti-dependency

```
for( i = 0; i < 1000; i++ )
{
    sum += a[i];
}
for( j = 0; j < 1000; j++ )
{
    prod *= a[j];
}
```

Adjacency

- Loops should not have any instructions between them

```
for( i = 0; i < 10; i++ )
{
    a[i] = b[i];
}

a[7] = 7;

for( i = 0; i < 10; i++ )
{
    c[i] = a[i];
}
```

Terminating Condition and Limit

- The terminating conditions and limits of both loops should be the same

```
for( i = 0; i < 100; i++ )
{
    sum += a[i];
}

for( i = 100; i < 500; i++ )
{
    prod *= a[i];
}
```

Data Dependency

- Exists between 2 statements if they access the same memory and one of them is a store
- Types of dependency:
  - True Dependency (Flow, Read After Write)
  - Anti Dependency (Write After Read)
  - Output Dependency (Write After Write)

Types of Dependency

True (RAW)

```
int a, b, c;

b = a;
c = b;
```

Anti (WAR)

```
int a, b;

a = b + 1;
b = 7;
```

Output (WAW)

```
int a, b, c;

b = 3;
a = b + 1;
b = 7;
```

Types of Dependency

Loop-Carried True Dependency

```
for( i = 0; i < 2; i++ )
{
    a[i+1] = b[i];
}

for( i = 0; i < 2; i++ )
{
    c[i] = a[i];
}
```

```
//Before Fusion

a[1]=b[0]
a[2]=b[1]
a[3]=b[2]

c[0]=a[0]
c[1]=a[1]
c[2]=a[2]
```

```
for( i = 0; i < 2; i++ )
{
    a[i+1] = b[i];
    c[i] = a[i];
}
```

```
//After Fusion

a[1]=b[0]
c[0]=a[0]

a[2]=b[1]
c[1]=a[1]

a[3]=b[2]
c[2]=a[2]
```

Types of Dependency

Loop-Carried Output Dependency

```
for( i = 0; i < 2; i++ )
{
    a[i+1] = 0;
}

for( i = 0; i < 2; i++ )
{
    a[i] = i;
}
```

```
//Before Fusion

a[1] = 0
a[2] = 0
a[3] = 0

a[0] = 0
a[1] = 1
a[2] = 2
```

```
for( i = 0; i < 2; i++ )
{
    a[i+1] = 0;
    a[i] = i;
}
```

```
//After Fusion

a[1] = 0
a[0] = 0

a[2] = 0
a[1] = 1

a[3] = 0
a[2] = 2
```

Types of Dependency

Loop-Carried Anti Dependency

```
for( i = 0; i < 2; i++ )
{
    a[i] = b[i];
}

for( i = 0; i < 2; i++ )
{
    c[i] = a[i+1];
}
```

```
//Before Fusion

a[0] = b[0];
a[1] = b[1];
a[2] = b[2];

c[0] = a[1];
c[1] = a[2];
c[2] = a[3];
```

```
for( i = 0; i < 2; i++ )
{
    a[i] = b[i];
    c[i] = a[i+1];
}
```

```
//After Fusion

a[0] = b[0];
c[0] = a[1];

a[1] = b[1];
c[1] = a[2];

a[2] = b[2];
c[2] = a[3];
```

Implementation in LLVM

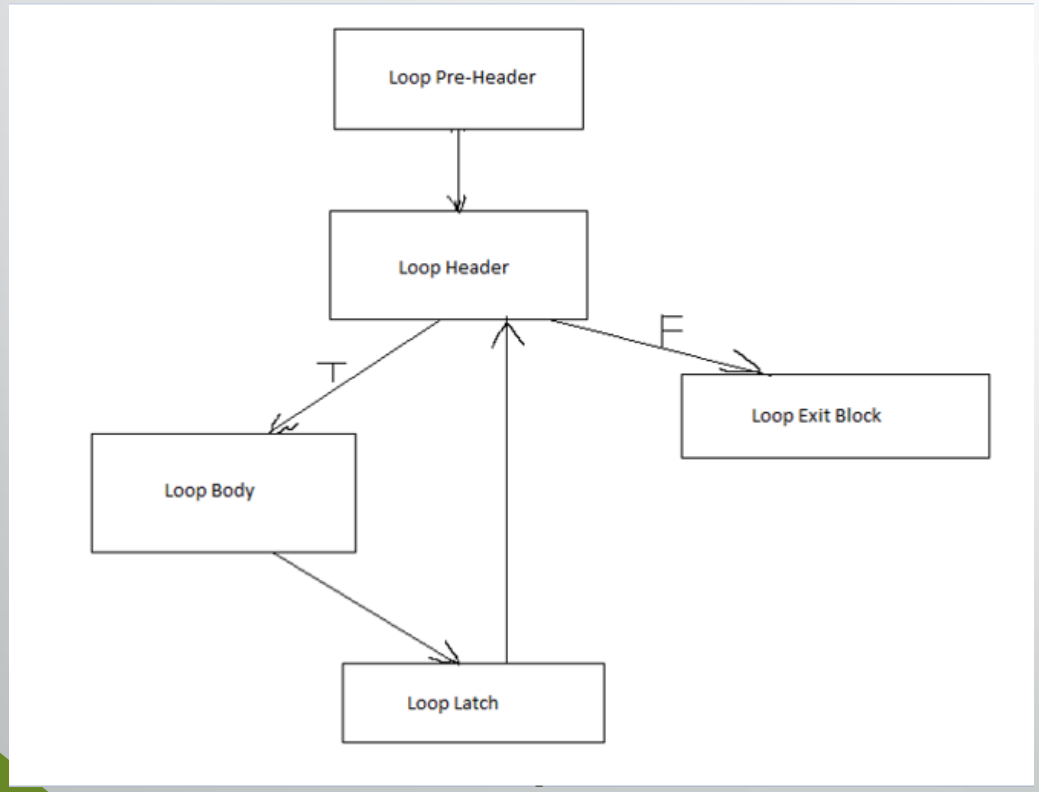
- LLVM = Low Level Virtual Machine
- It takes Intermediate Representation (IR) from the front end and emits the Optimized IR
- Iterate over all functions and fuse loops in them

Prerequisites for Fusion

There are 2 main prerequisites before attempting fusion

- loop–simplify
  - Simplified loops to a general structure
- indvars
  - Tried to Canonize the induction variable

loop–simplify



- This is the structure of loops handled in compilers (GCC, Open64, LLVM)
- The pass tries to convert the loops in IR to this form

Indvars

- Guarantees that the induction variable is a **Canonical Induction Variable**
- Initialized to O and incremented by 1

```
//Before indvars

for( i = 0;i < 100; i = i+10 )
    arr[i] = i ;
```

```
//After indvars

for( i = 0;i < 10; i++ )
    arr[i*10] = i*10 ;
```

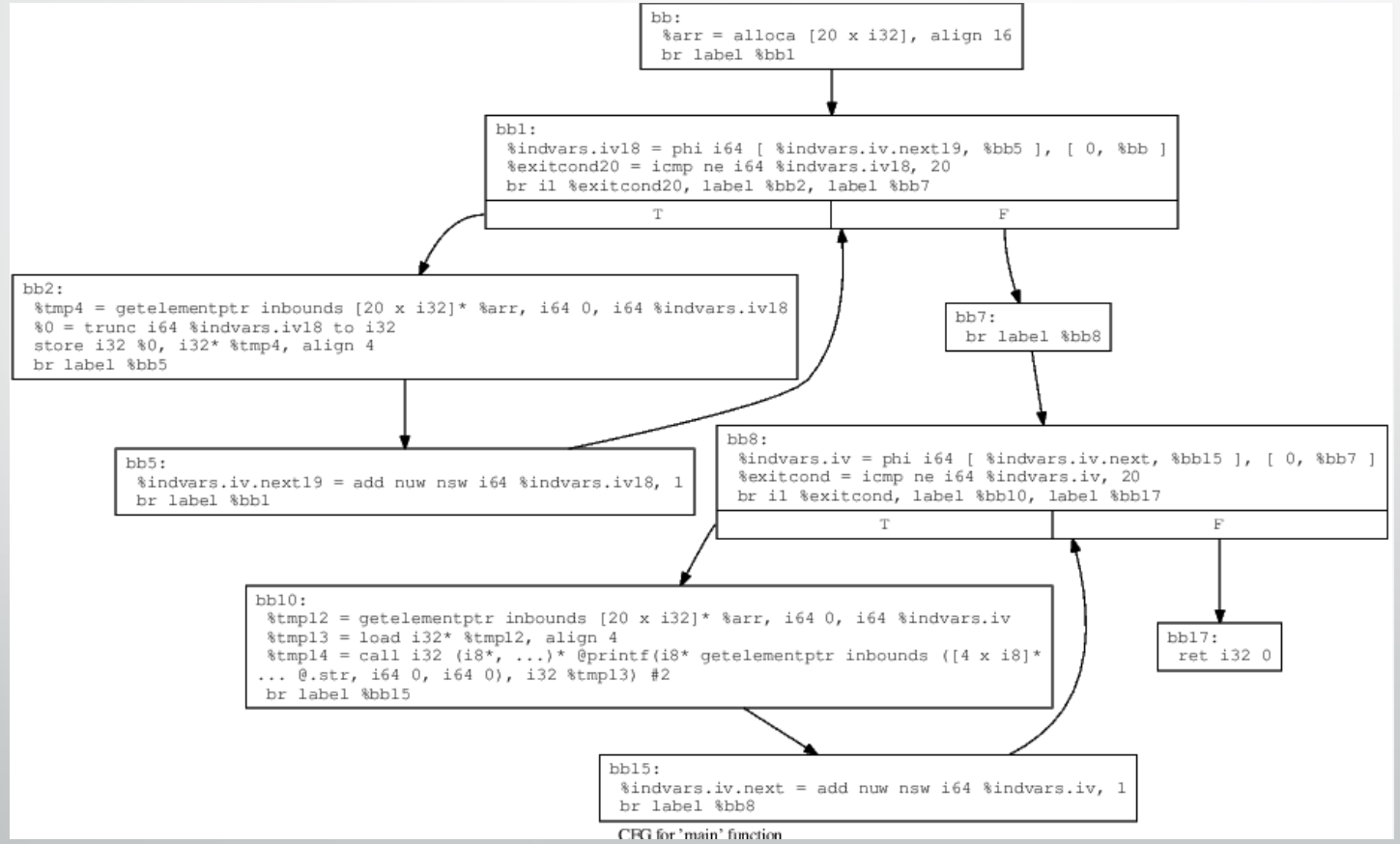
Fusing Two Loops:

Fusion in LLVM

Steps to fuse two loops–

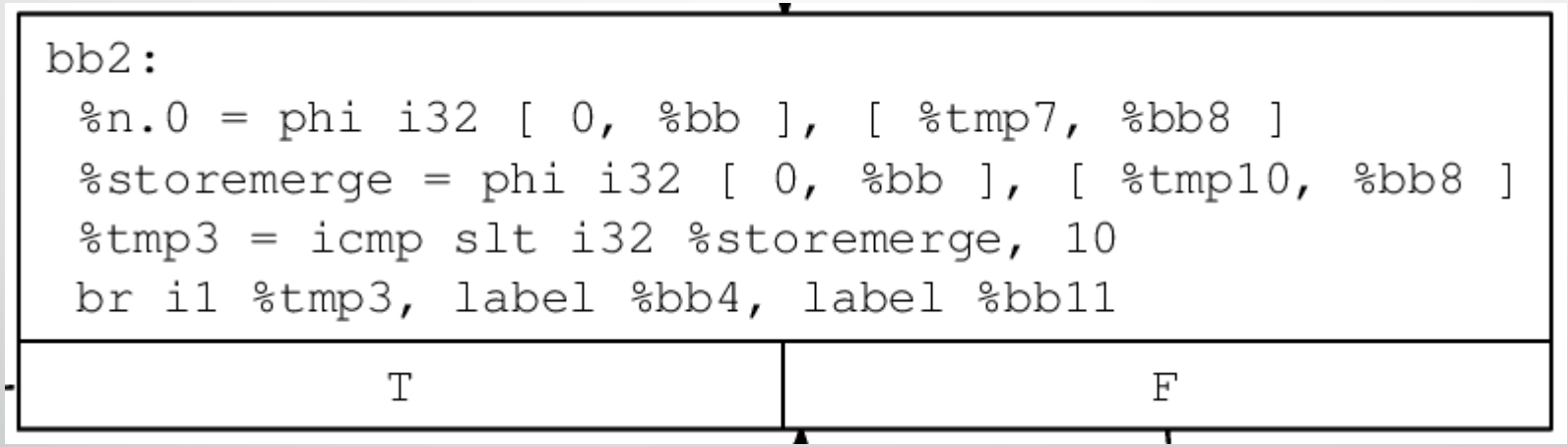
1. Basic Checks
    1. Checking adjacency
    2. Terminating condition & limiting value
2. Replace induction variable
3. Dependency check
4. Delete unwanted basic blocks

CFG Before Fusion



CFG for 'main' function

Basic Checks



Replace index variable

```
for( i = 0; i < 1000; i++ )
{
    sum += a[i];
    prod *= a[j];
}
```

Steps in checking dependency

1. Fuse the loops temporarily
2. Get LOAD and STORE instructions
3. Check for dependency between the two
4. If ANTI
    1. Revert back to original structure
    2. Replace induction variable
5. Else
    1. Delete Unwanted blocks

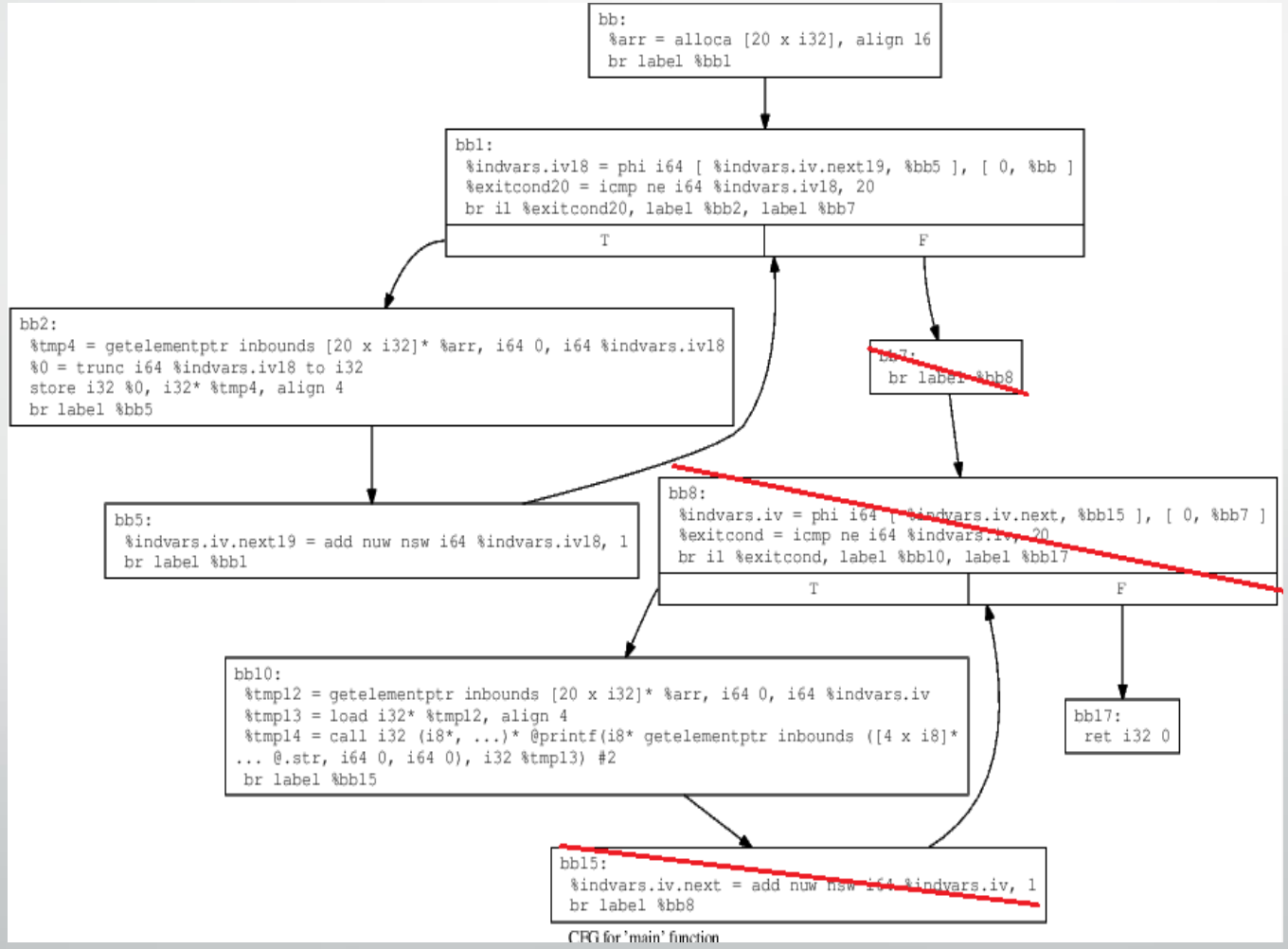
Why temporary fuse?

```
for( i = 0; i < 2; i++ )
{
    a[i] = b[i];
}

for( i = 0; i < 2; i++ )
{
    c[i] = a[i+1];
}
```

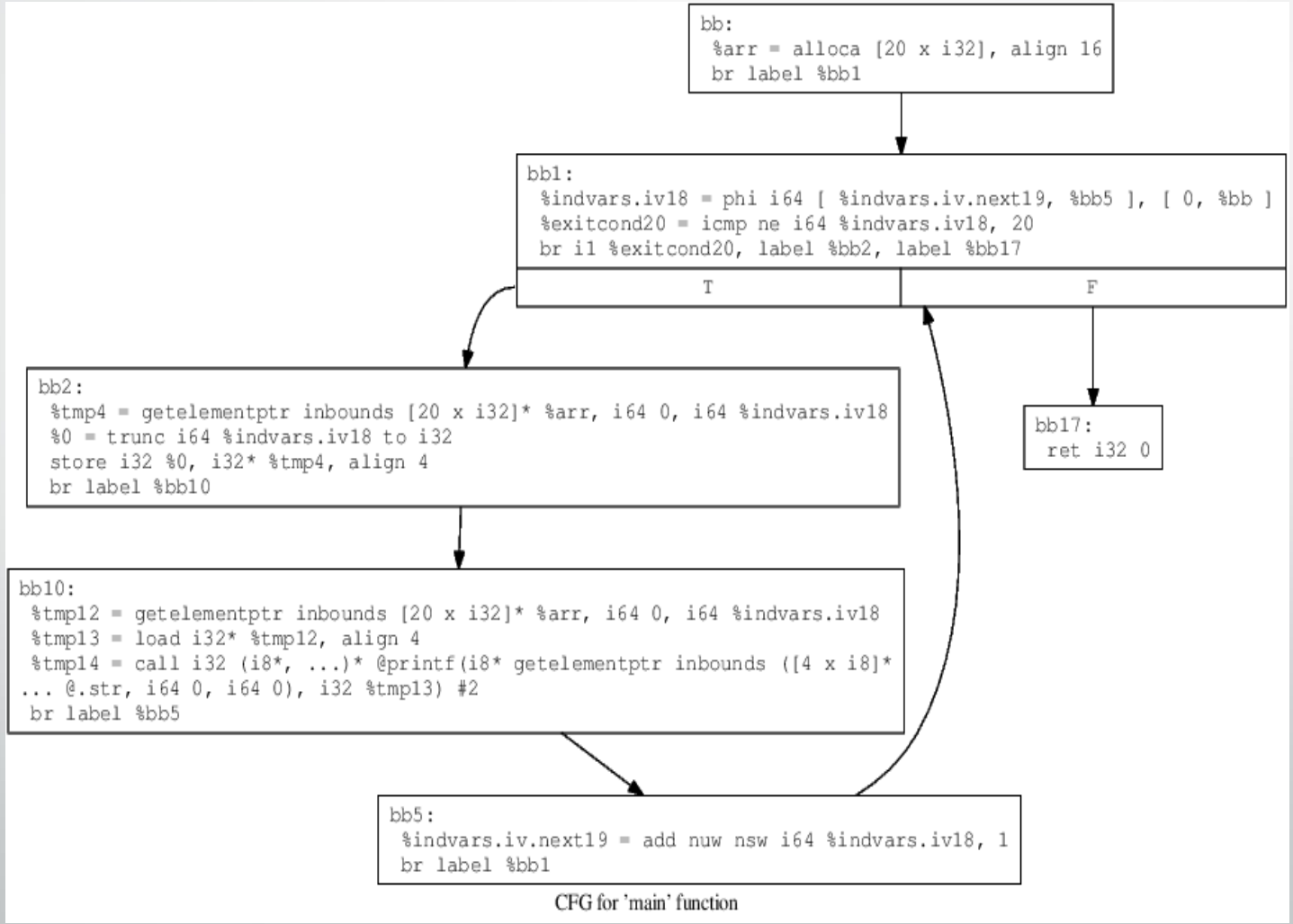
```
for( i = 0; i < 2; i++ )
{
    a[i] = b[i];
    c[i] = a[i+1];
}
```

Deleting Unwanted Blocks



CFG for 'main' function

CFG after fusion



CFG for 'main' function

Profitability

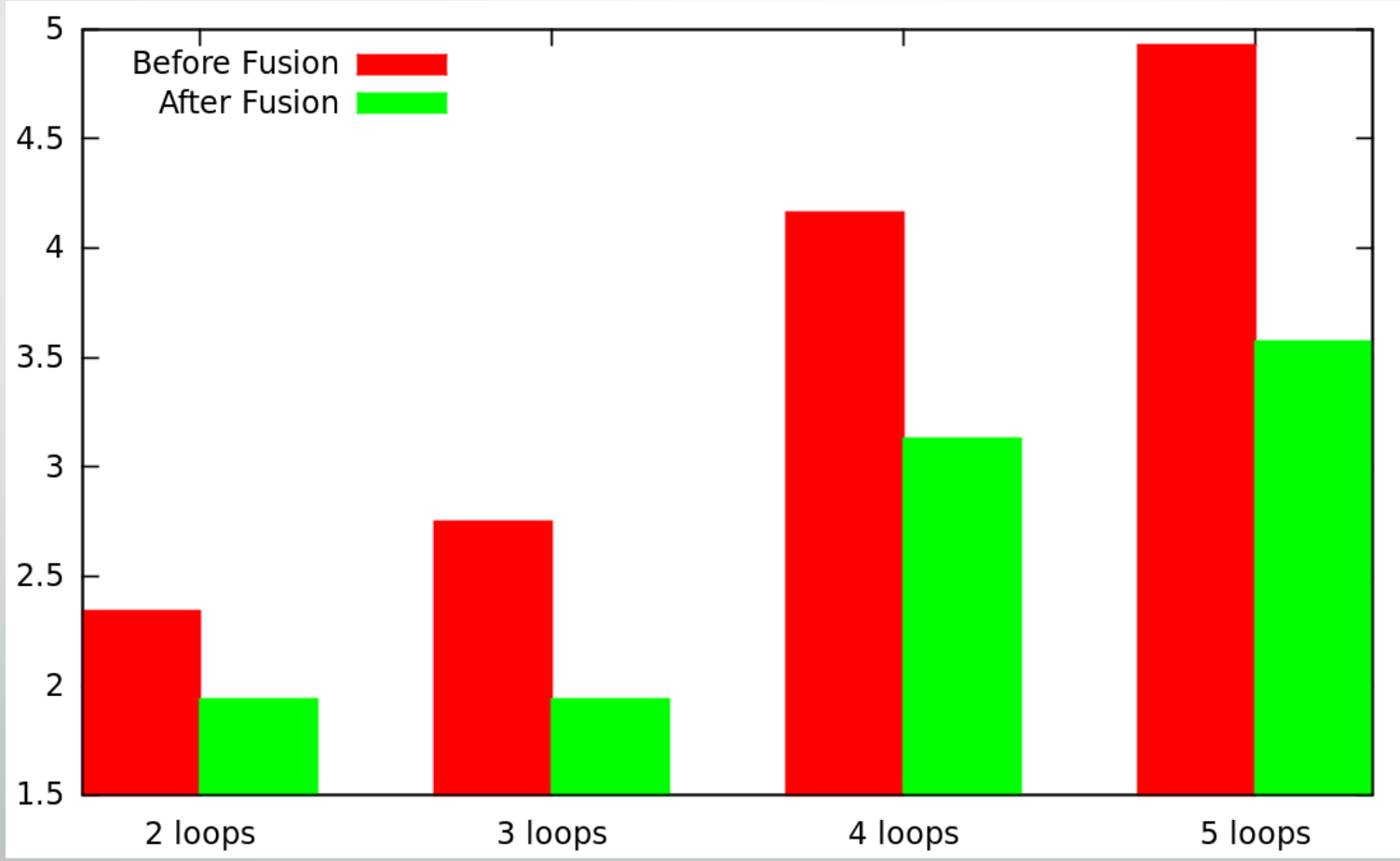
Why Profitability?

- Loop Fusion does not always increase runtime.
- Depends on two factors:
  - Extensive use of cache
    - Problem occurs when there are many arrays
  - Reuse of registers
    - Problem occurs when there are limited number of registers
- Many heuristics are available for checking profitability

Graph Construction

- Steps
  - Each loop is fused with every other loop
  - Dependency is computed
  - Node represents a loop
  - Edge represents dependency type and weight
- This gives way to analyze many heuristics

Results


Before Fusion
After Fusion
5
4.5
4
3.5
3
2.5
2
1.5
2 loops
3 loops
4 loops
5 loops

Future Work

- Use of other heuristics to calculate profitability with the help of the graph
- Fuse loops which are not adjacent
- Fuse loops in which the induction variable changes within the body of the loop

References

- Optimizing Compilers for Modern Architectures: A Dependence-Based Approach by Randy Allen and Ken Kennedy
- Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jerey D. Ullman
- The LLVM Compiler Infrastructure: http://www.llvm.org

Thank You ☺

Any Questions??