
Exploiting Speculative Thread-Level Parallelism in Data Compression Applications

Shengyue Wang, Antonia Zhai, and Pen-Chung Yew

**Department of Computer Science & Engineering
University of Minnesota**



UNIVERSITY OF MINNESOTA

Thread-Level Speculation (TLS)

- ✓ Facilitates automatic parallelization on Chip Multiprocessors (CMP) by:
 - Executing potentially dependent threads in parallel
 - Preserving data dependences via runtime checking
- ✗ Limited performance has been achieved for:
 - General-purpose applications such as SPEC CINT2000

Need a set of compiler optimizations



Why Compression Benchmarks?

[Wang_Icpc05]: Search for parallelism in loops

- Studied all loops in SPEC CINT2000 exhaustively
- Some of the loops selected for parallel execution show poor performance

This paper: Develop new compiler optimizations

- Instruction scheduling
- Reduction transformation
- Iteration merging

Compression benchmarks:

- Significant coverage of loops
- Good performance potential



Compression Benchmarks

GZIP

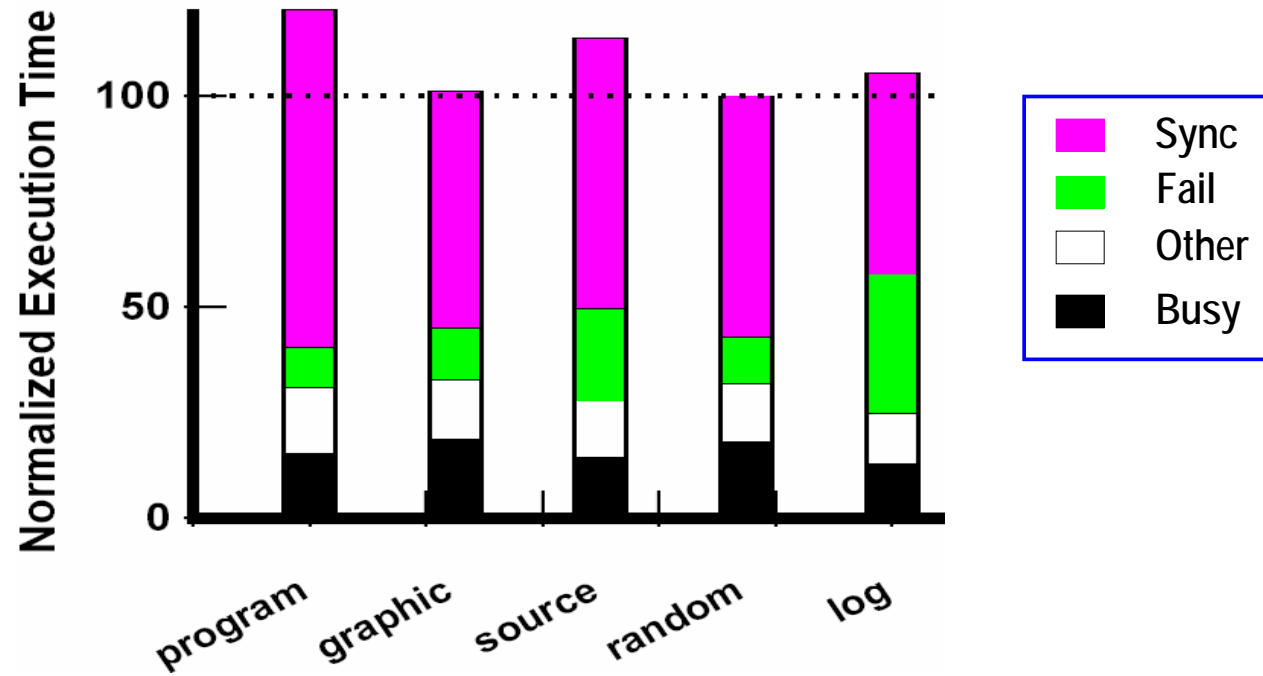
- Dictionary-based algorithm
- Hash table is used to detect repeated strings

BZIP2

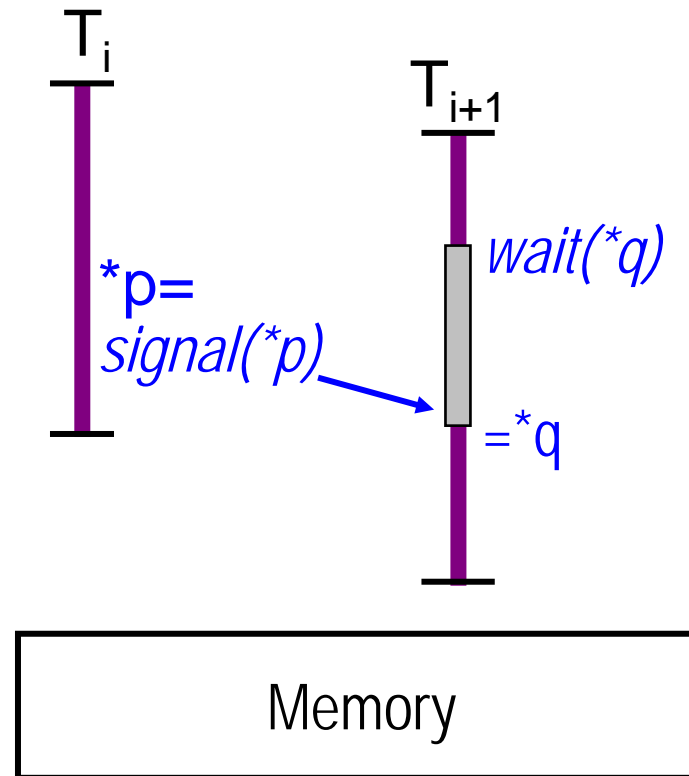
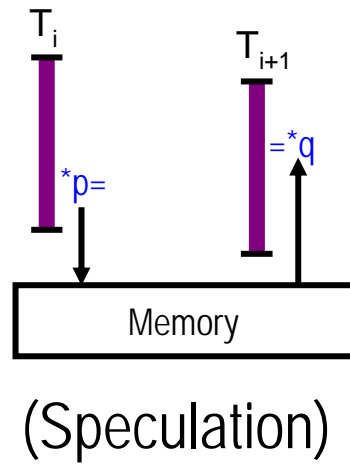
- Block-based algorithm (block size: 100k-900k)
- Burrows-wheeler transform on each block (sorting)



Program Performance of GZIP

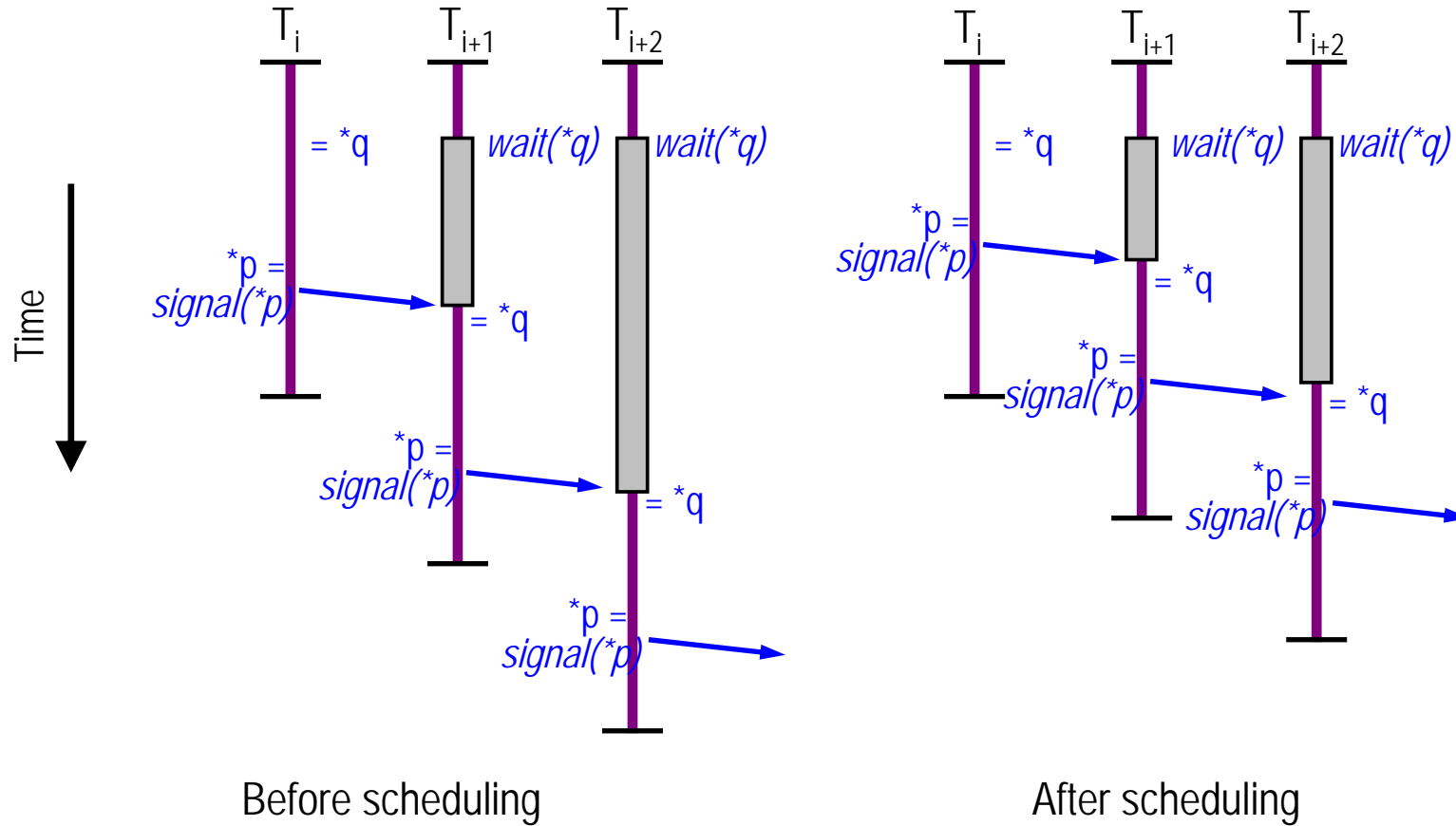


Synchronization



good when $p == q$

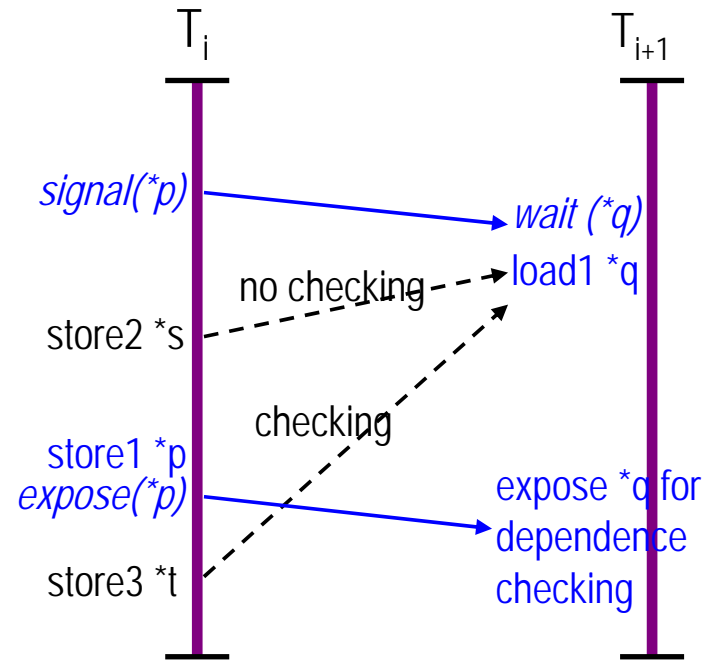
Instruction Scheduling



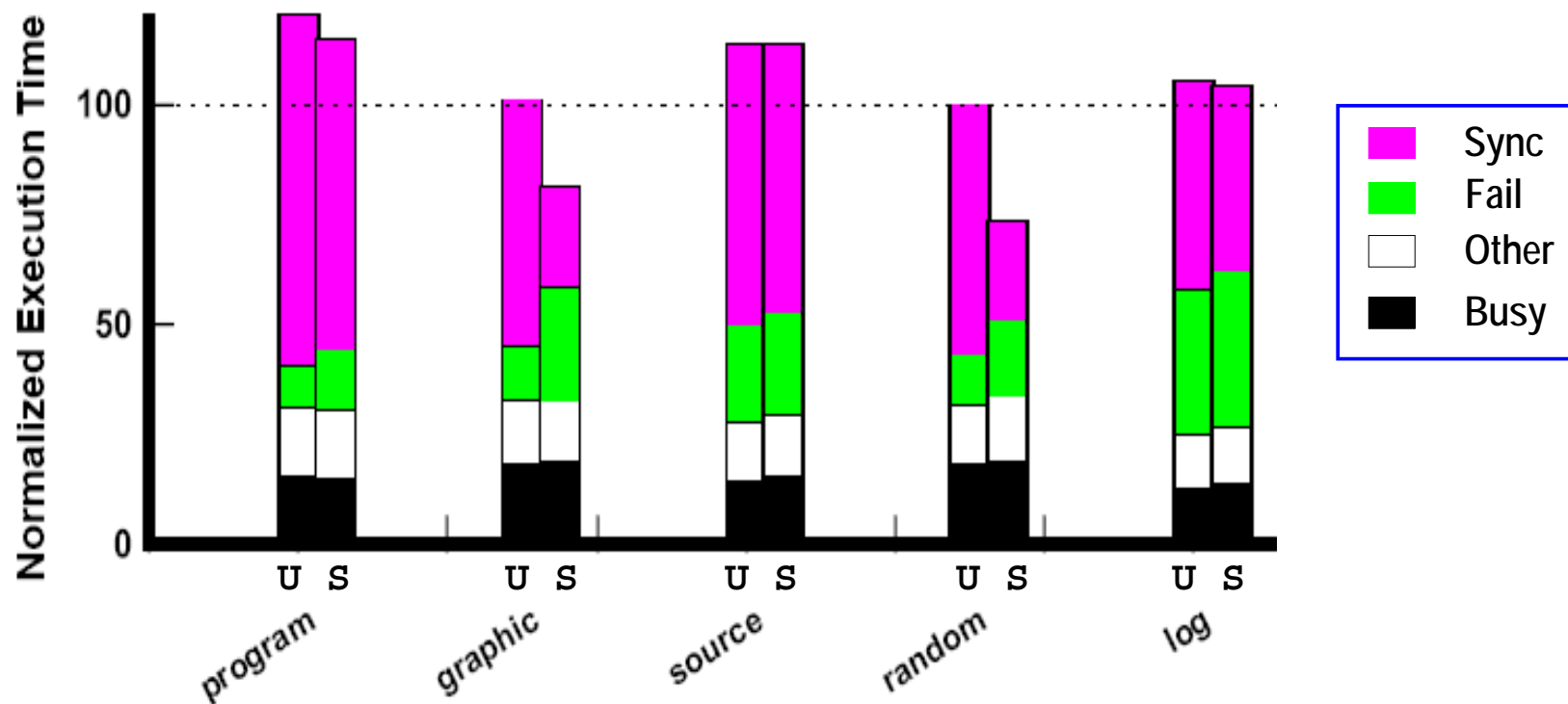
Aggressive Instruction Scheduling

- *Intra-thread* data/control speculation
 - Allow more aggressive instruction scheduling
 - Speculatively compute and forward data to the consumer thread
- Recovery code support
 - Reduce the *intra-thread* mis-speculation cost
 - Re-compute and re-forward data to the consumer thread

How Synchronization and Speculation Work Together?

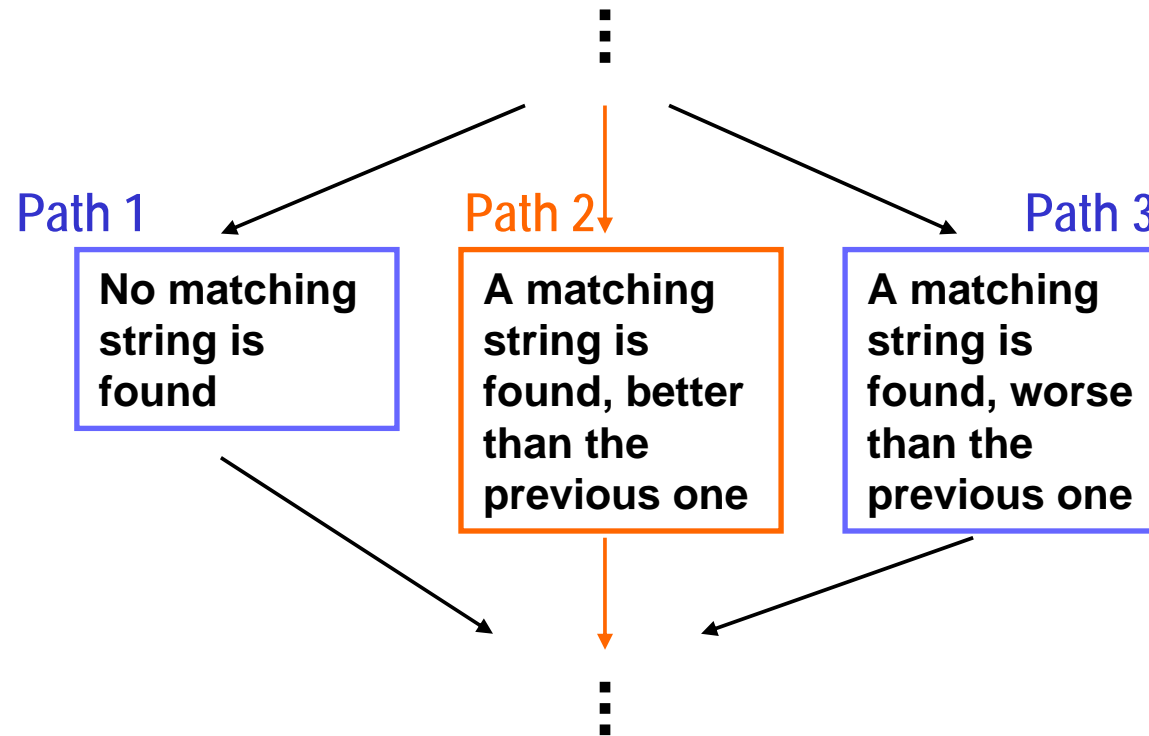


Impact of Instruction Scheduling



U = Un-optimized
S = Instruction scheduling

Input Sensitivity in GZIP



Three possible paths in the outer loop in deflate(), and path 2 is identified as the most frequent path.

Compression Applications

GZIP

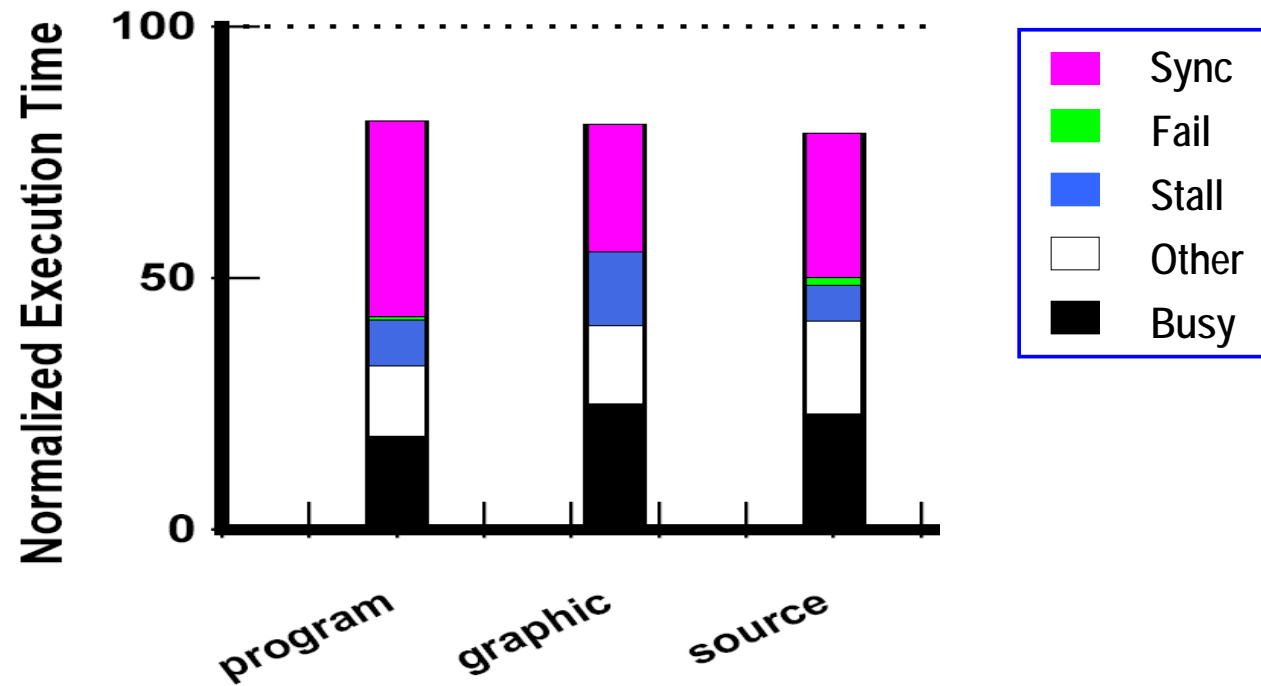
- Dictionary-based algorithm
- Hash table is used to detect repeated strings

BZIP2

- Block-based algorithm (block size: 100k-900k)
- Burrows-wheeler transform on each block (sorting)



Program Performance of BZIP2



Traditional Reduction Elimination

```
while(cond1) {  
  while(cond2) {  
    sum++;  
  }  
}
```

A reduction variable

```
while(cond1) {  
  wait(sum);  
  while(cond2) {  
    sum++;  
  }  
  signal(sum);  
}
```

Synchronization
(outer loop is parallelize)

```
while(cond1) {  
  while(cond2) {  
    sum[j]++;  
  }  
}  
  
while(cond1) {  
  sum+=sum[j];  
}
```

Reduction elimination



Reduction Transformation

```
while(cond1) {  
  while(cond2) {  
    sum++;  
  }  
  ...  
  =sum;  
  ...  
}
```

Use of immediate
result of sum

```
while(cond1) {  
  sum0 = 0;  
  while(cond2) {  
    sum0++;  
  }  
  ...  
  wait(sum);  
  =sum+sum0;  
  ...  
  sum+=sum0;  
  signal(sum);  
}
```

Sum0 is thread private



Reduction Variable in BZIP2

- Reduction variable “workDone” is used in the bucket sort loop in Burrows-Wheeler Transform
 - Count the number of total comparisons
 - Test in each loop iteration to avoid too many comparisons
- The test is rarely true, and the outcome is highly predictable



Speculative Reduction Transformation

```
while(cond1) {  
    wait(sum);  
    if(sum+sum0>x)  
        work1;  
    else  
        work2;  
    ...  
    while(cond2) {  
        sum0++;  
    }  
    ...  
    sum+=sum0;  
    signal(sum);  
}
```

Use of reduction variable to determine branch outcome

```
while(cond1) {  
    sum0'=sum0; x'=x;  
    work2;  
    ...  
    while(cond2) {  
        sum0++;  
    }  
    ...  
    wait(sum);  
    if(sum+sum0'>x') {  
        recovery;  
    }  
    sum+=sum0;  
    signal(sum);  
}
```

Speculative reduction transformation



Speculative Reduction Transformation

```
while(cond1) {  
    wait(sum);  
    while(cond2) {  
        sum0++;  
        if (sum+sum0>100)  
            return;  
        work1;  
    }  
    ...  
    sum+=sum0;  
    signal(sum);  
}
```

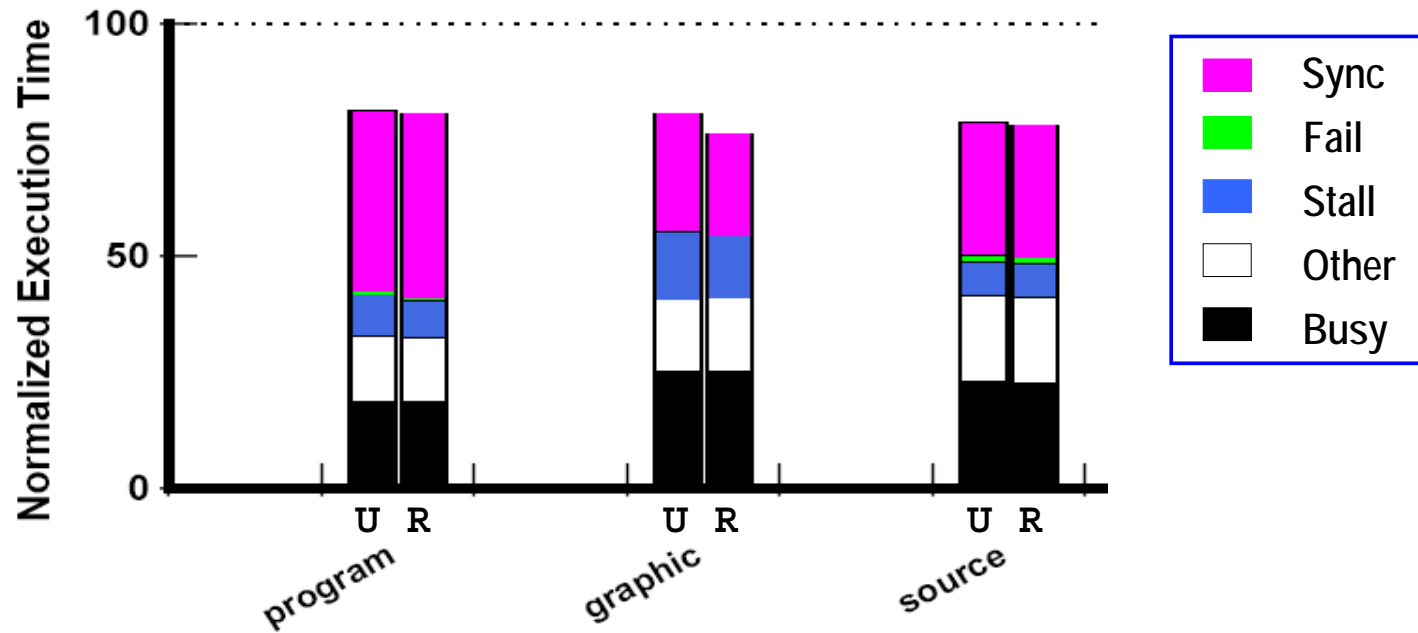
Use of reduction variable in
the inner loop

```
while(cond1) {  
    while(cond2) {  
        sum0++;  
        work1;  
    }  
    ...  
    wait(sum);  
    if (sum+sum0>100) {  
        recovery;  
    }  
    sum+=sum0;  
    signal(sum);  
}
```

Speculative reduction
transformation



Impact of Reduction Transformation

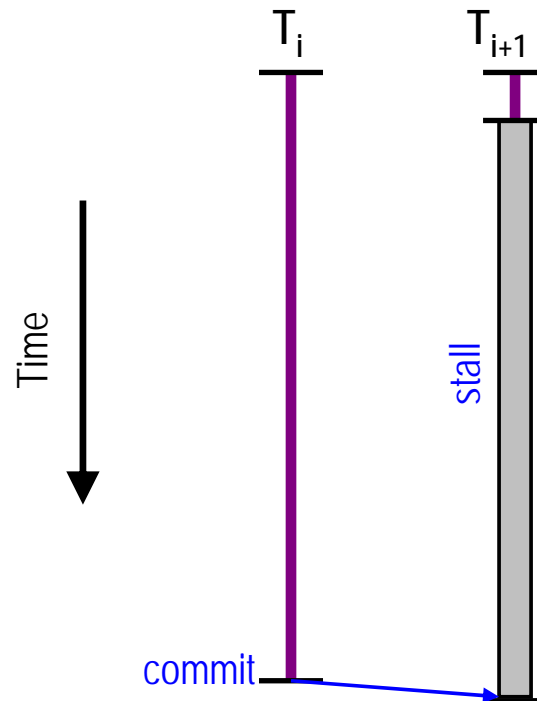


U = Un-optimized

R = Reduction transformation



“Stall” Caused by Unbalanced Workloads

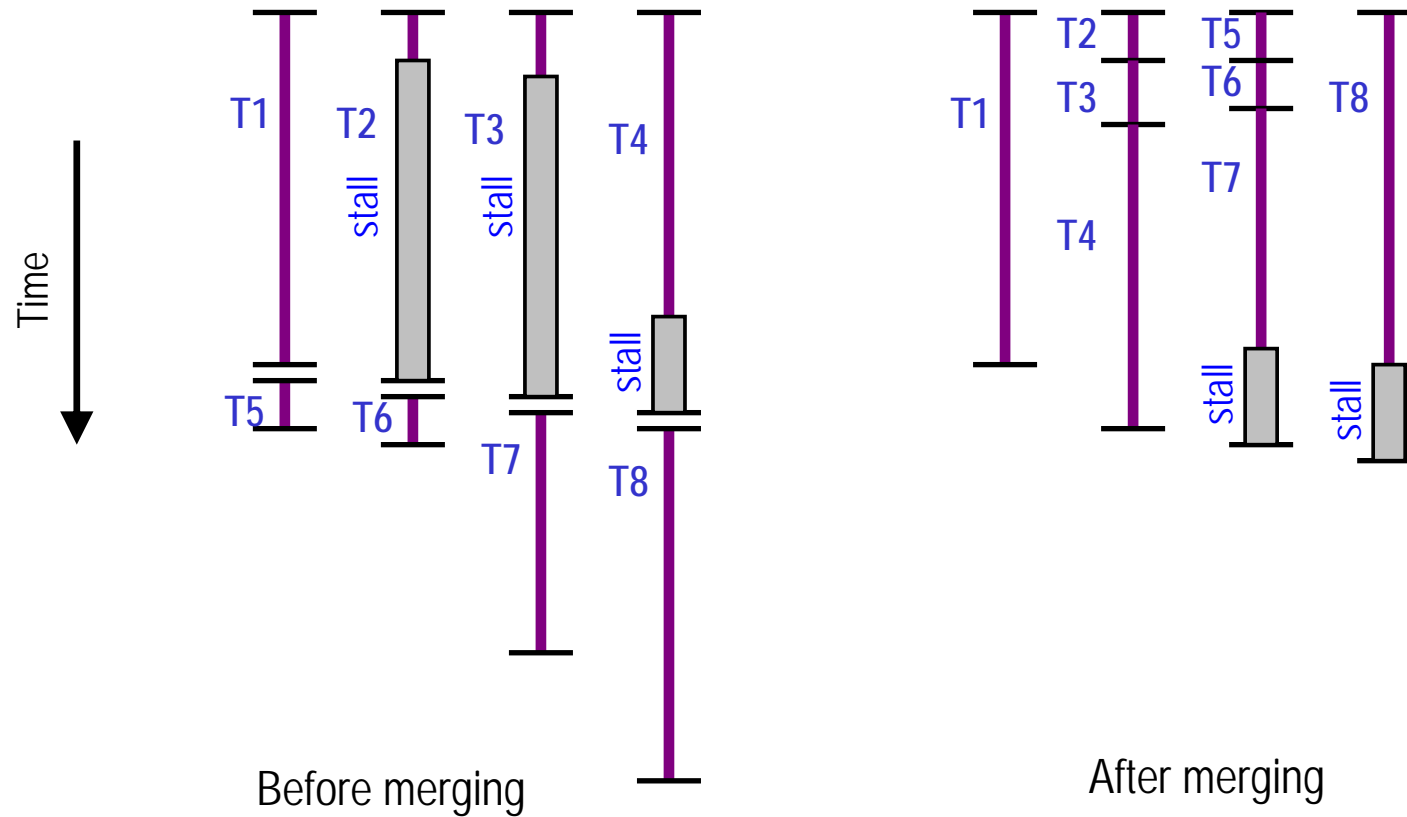


Unbalanced Workloads in BZIP2

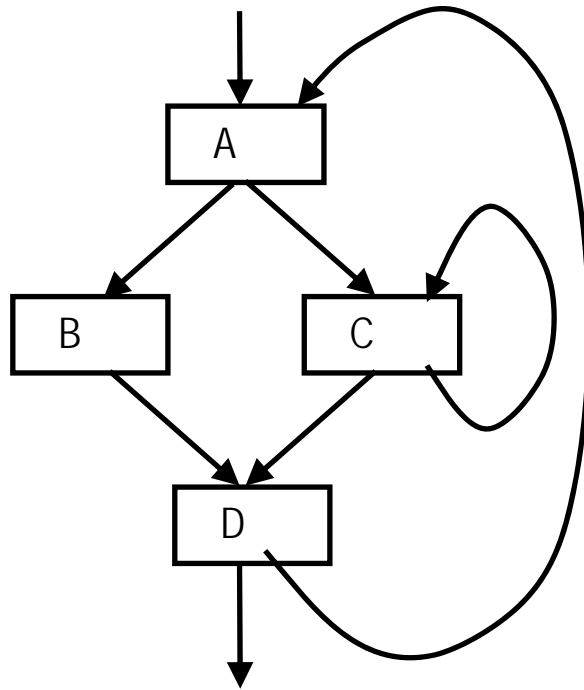
- Bucket sort loop in Burrows-Wheeler Transform
 - Short iteration: if a bucket is sorted, do nothing
 - Long iteration: if unsorted, sort it by calling quick sort
- Significant stall due to the variance in thread size



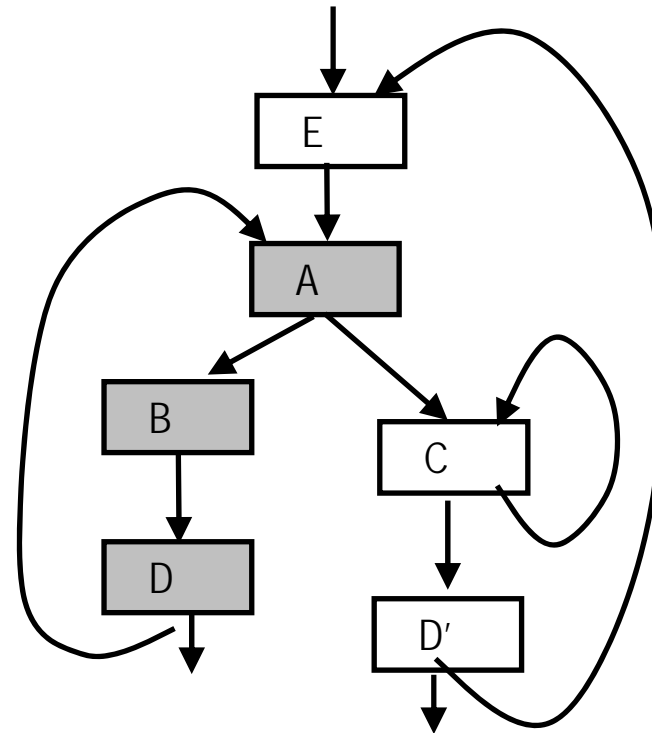
Iteration Merging



Transformation for Iteration Merging



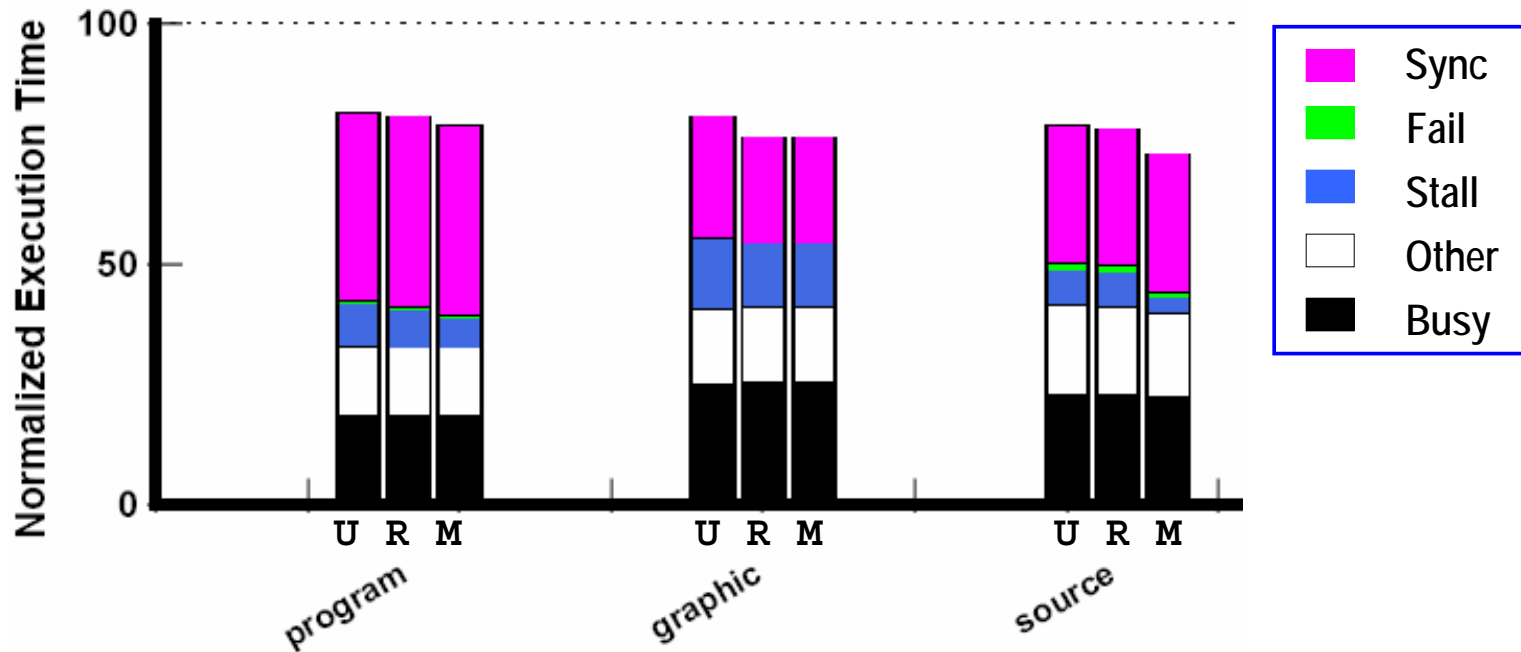
Before transformation



After transformation



Impact of Iteration Merging



U = Un-optimized
R = Reduction transformation
M = Iteration merging

Conclusions

Developed three new compiler optimizations:

- GZIP
 - Instruction scheduling (up to 36%)
- BZIP2
 - Reduction transformation (up to %7)
 - Iteration merging (up to %9)

We need a large set of optimizations targeting different program behaviors to fully exploit the potential of TLS



Thank You!