Speculative Code Compaction:

Eliminating Dead Code via Speculative Microcode Transformations

Logan Moody, Wei Qi, Abdolrasoul Sharifi, Layne Berry, Joey Rudek, Jayesh Gaur, Jeff Parkhurst, Sreenivas Subramoney, Kevin Skadron, Ashish Venkat

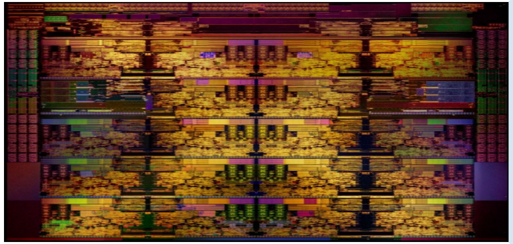
NSF
UNIVERSITY of VIRGINIA
intel

Speculative Code Compaction

- Motivation
- Overview of the Framework
- Results
- Conclusion

UNIVERSITY of VIRGINIA

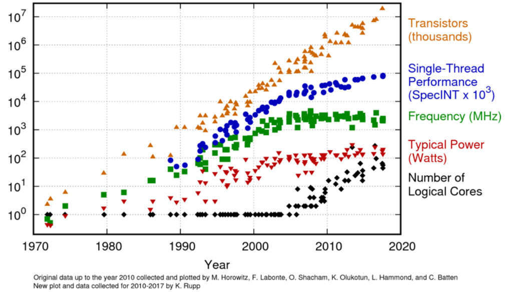
The Landscape of Modern Computing

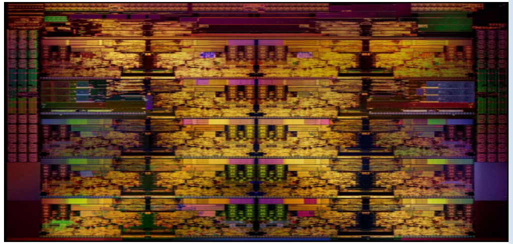


Hardware
(high core counts, specialized cores)

Software
(rapidly evolving, increasingly complex)

The Landscape of Modern Computing





Hardware
(stagnation of single thread performance)

Software
(a substantial chunk of our workloads is inherently sequential)

University of Virginia

The Widening Hardware-Software Gap

Despite advances in compiler technology, a considerable chunk of wasteful computation still persists even in highly machine-tuned code.

```
i = 0;
while (i < n) {
        a = 5;
        if (a > 0) {
                sum += a;
                i++;
        } else {
                i += 2;
        }
}
```

→

```
sum += 5*n;
```

Optimizable at compile-time

The Widening Hardware-Software Gap

Despite advances in compiler technology, a considerable chunk of wasteful computation still persists even in highly machine-tuned code.

```
i = 0;
while (i < n) {
        a = x[i];
        if (a > 0) {
                sum += a;
                i++;
        } else {
                i += 2;
        }
}
```

Not optimizable at compile-time

The Widening Hardware-Software Gap

Despite advances in compiler technology, a considerable chunk of wasteful computation still persists even in highly machine-tuned code.

```
i = 0;
while (i < n) {
        a = x[i];
        if (a > 0) {
                sum += a;
                i++;
        } else {
                i += 2;
        }
}
```

Not optimizable at compile-time

But what if the values of array x are predictable at run-time?

University of Virginia

The Widening Hardware-Software Gap

Despite advances in compiler technology, a considerable chunk of wasteful computation still persists even in highly machine-tuned code.

```
i = 0;
while (i < n) {
        a = x[i];
        if (a > 0) {
                sum += a;
                i++;
        } else {
                i += 2;
        }
}
```

Optimization 1

```
i = 0;
sum = 0;
while (i < n) {
        sum += x[i];
        i++;
}
```

Optimization 2

```
i = 0;
sum = 0;
i = 2*((n+1)/2)
```

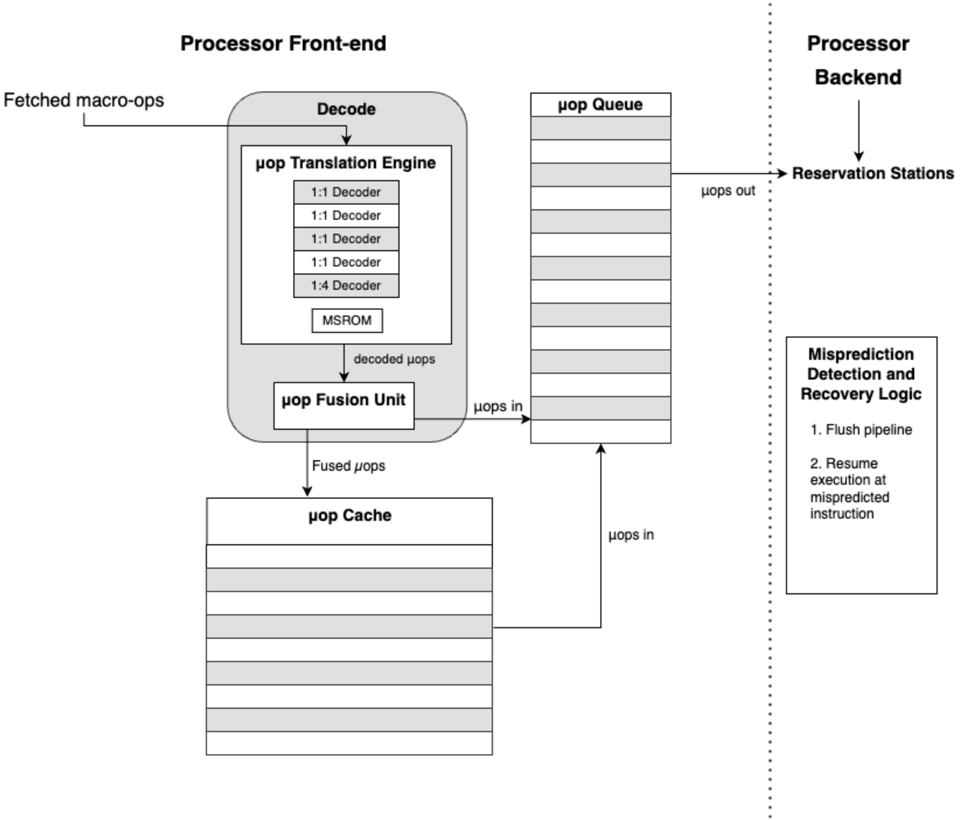
Not optimizable at compile-time

But what if the values of array x are predictable at run-time?

Speculative Code Compaction

UNIVERSITY of VIRGINIA

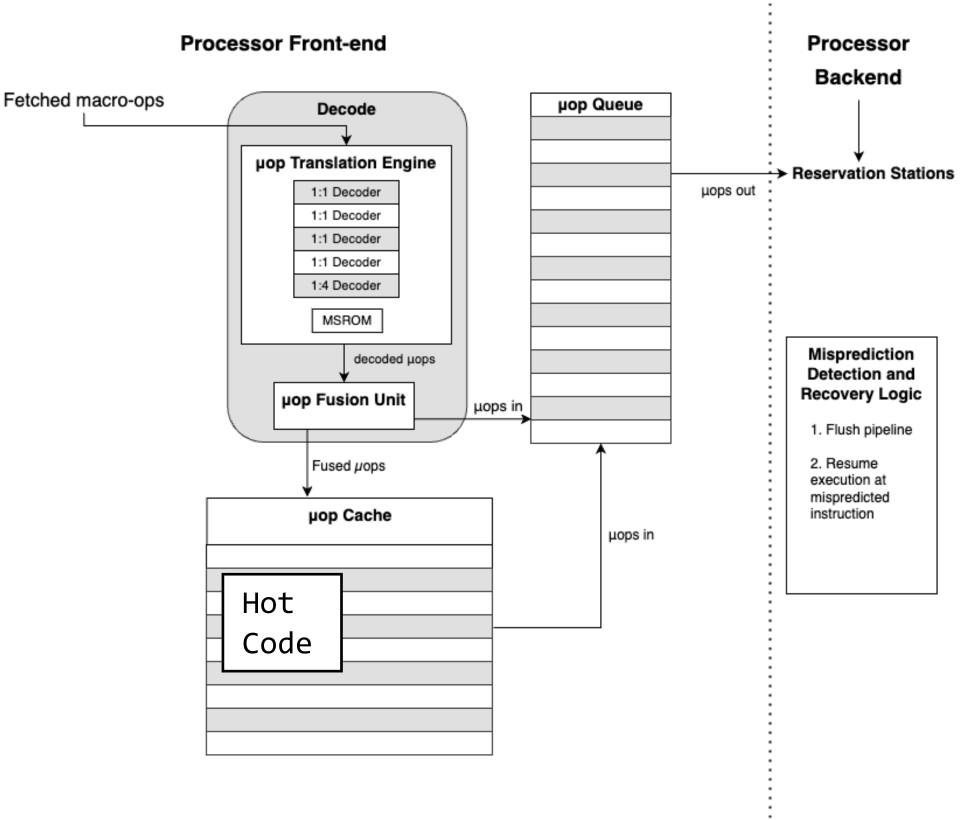
Speculative Code Compaction



Intel Front-end

Legacy Decode and µop Cache

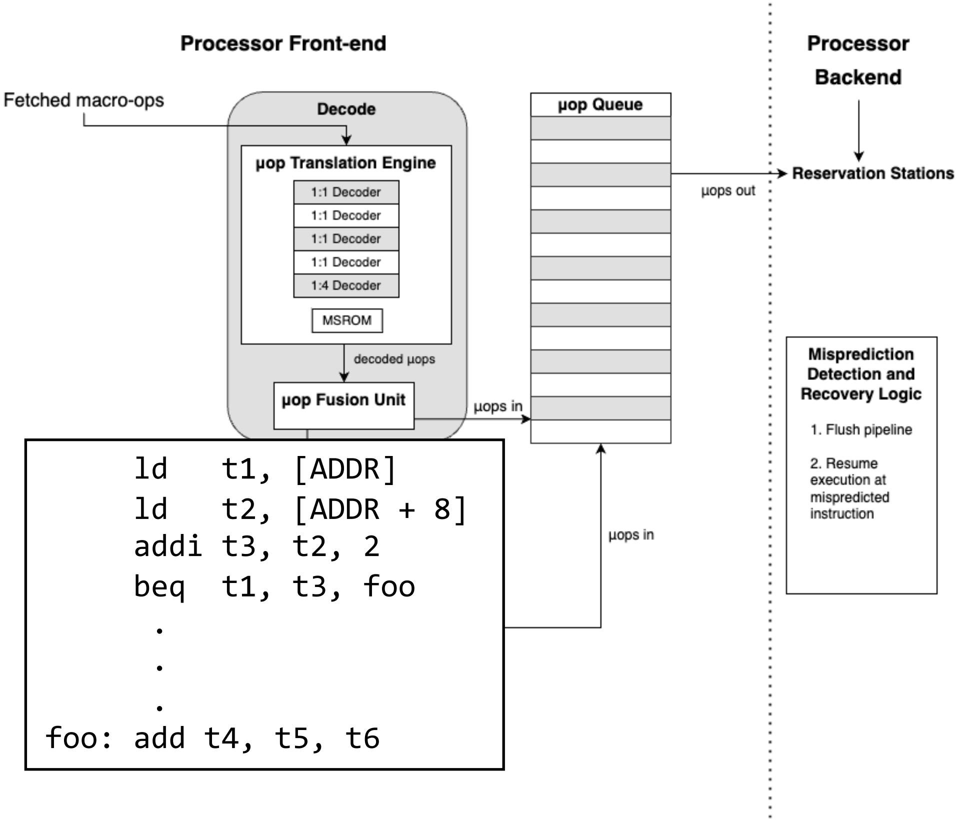
Speculative Code Compaction



Step 1: Hot Code Detection

Identify regions of hot code in µop cache

Speculative Code Compaction

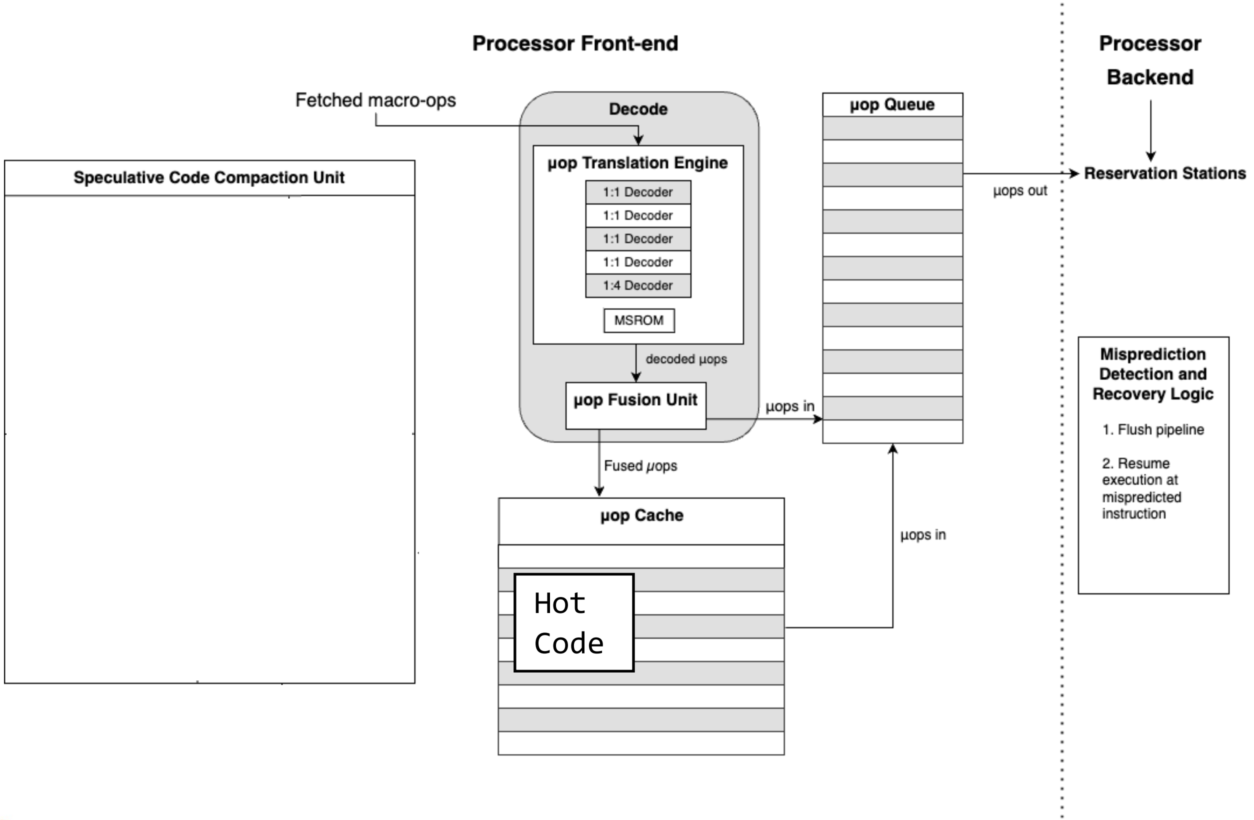



```
     ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
       .
       .
       .
foo: add t4, t5, t6
```

Step 1: Hot Code Detection

Identify regions of hot code in µop cache

Speculative Code Compaction

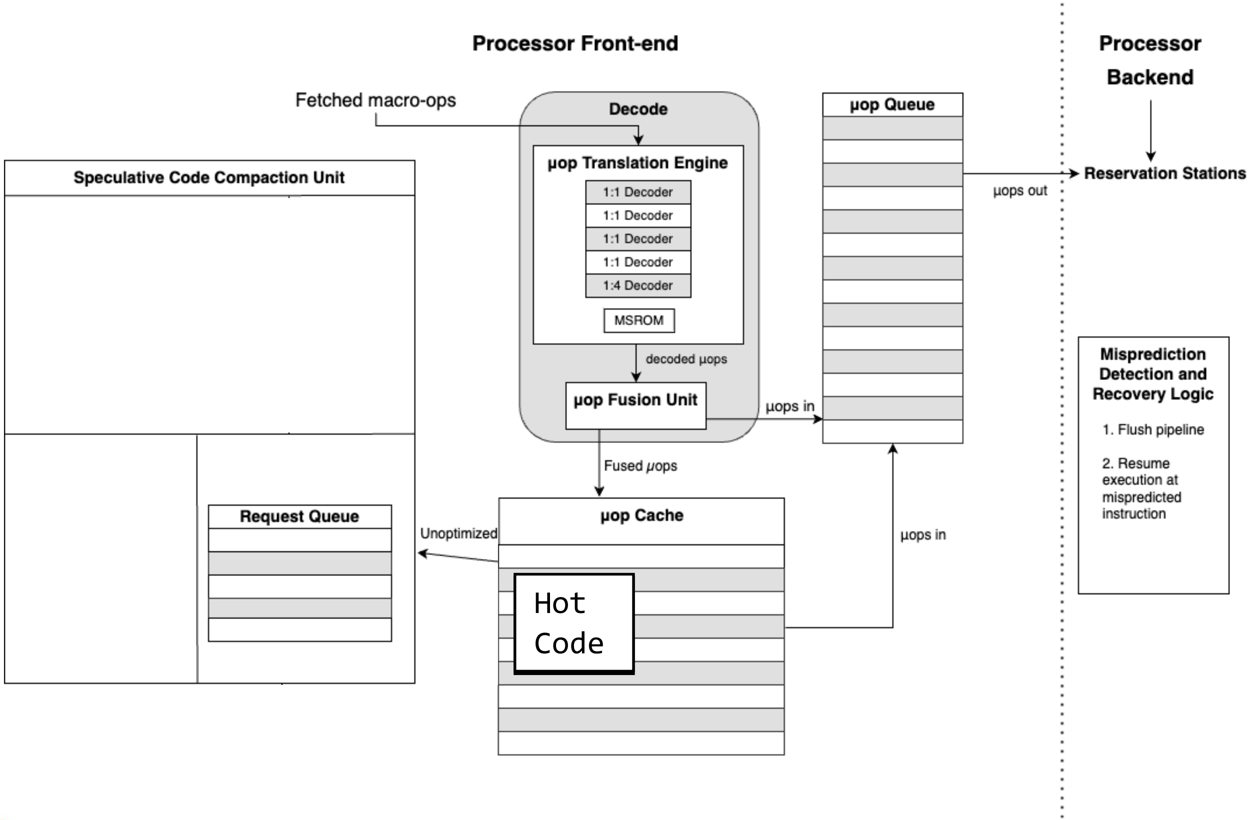


Step 1: Hot Code Detection

Identify regions of hot code in µop cache

```
     ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
      .
      .
      .
foo: add t4, t5, t6
```

Speculative Code Compaction

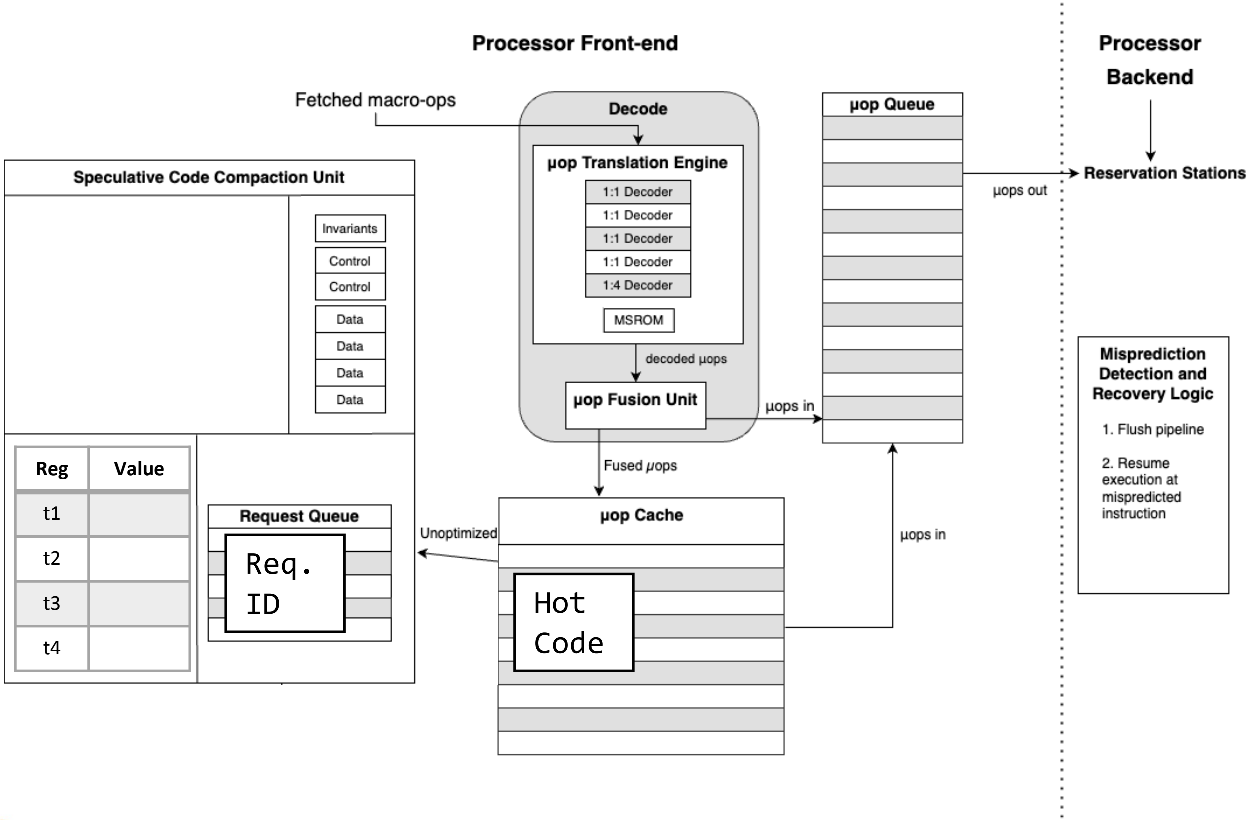


Step 2: Generate Request for Hot Code Region

Request Optimization from Code Compaction Unit

```
     ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
      .
      .
      .
foo: add t4, t5, t6
```

Speculative Code Compaction



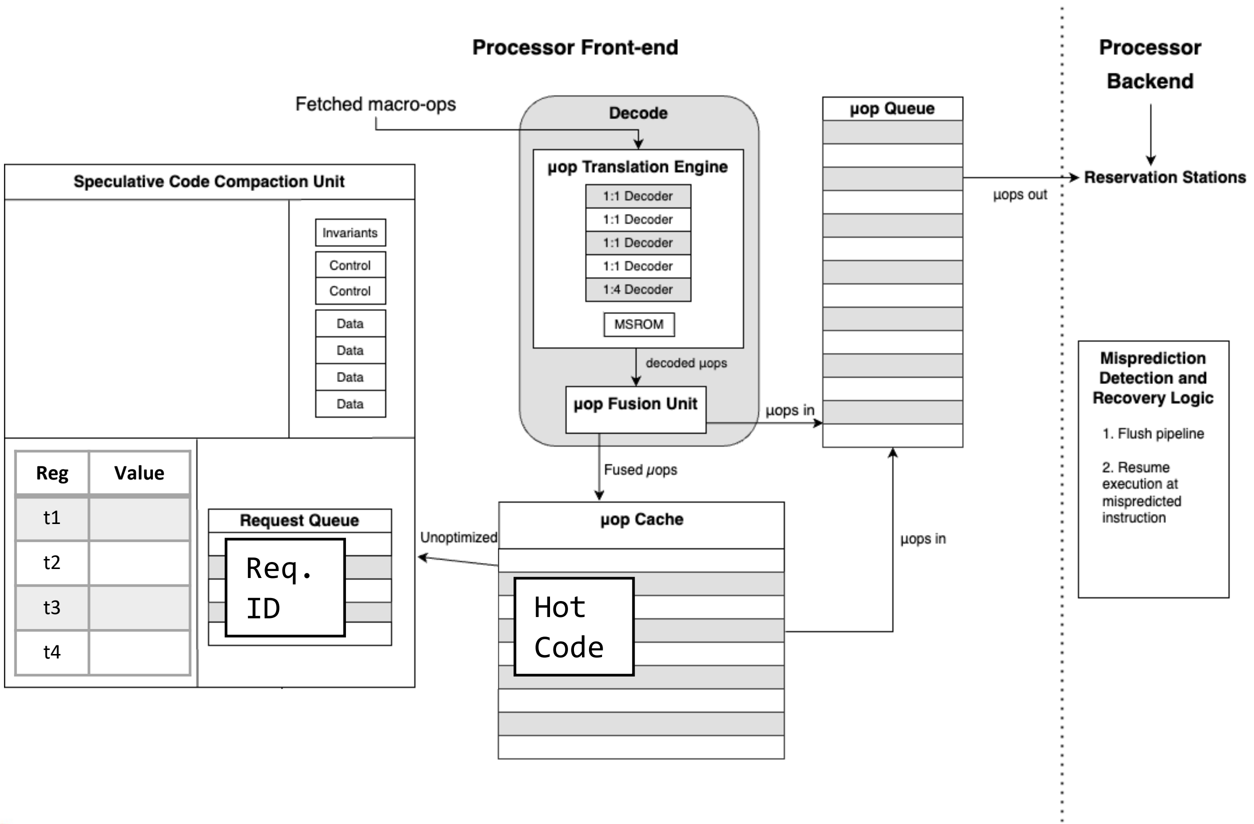
Step 3: Perform Optimizations

Track register context and prediction sources

Process one µop per cycle

```
     ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
       .
       .
       .
foo: add t4, t5, t6
```

Speculative Code Compaction

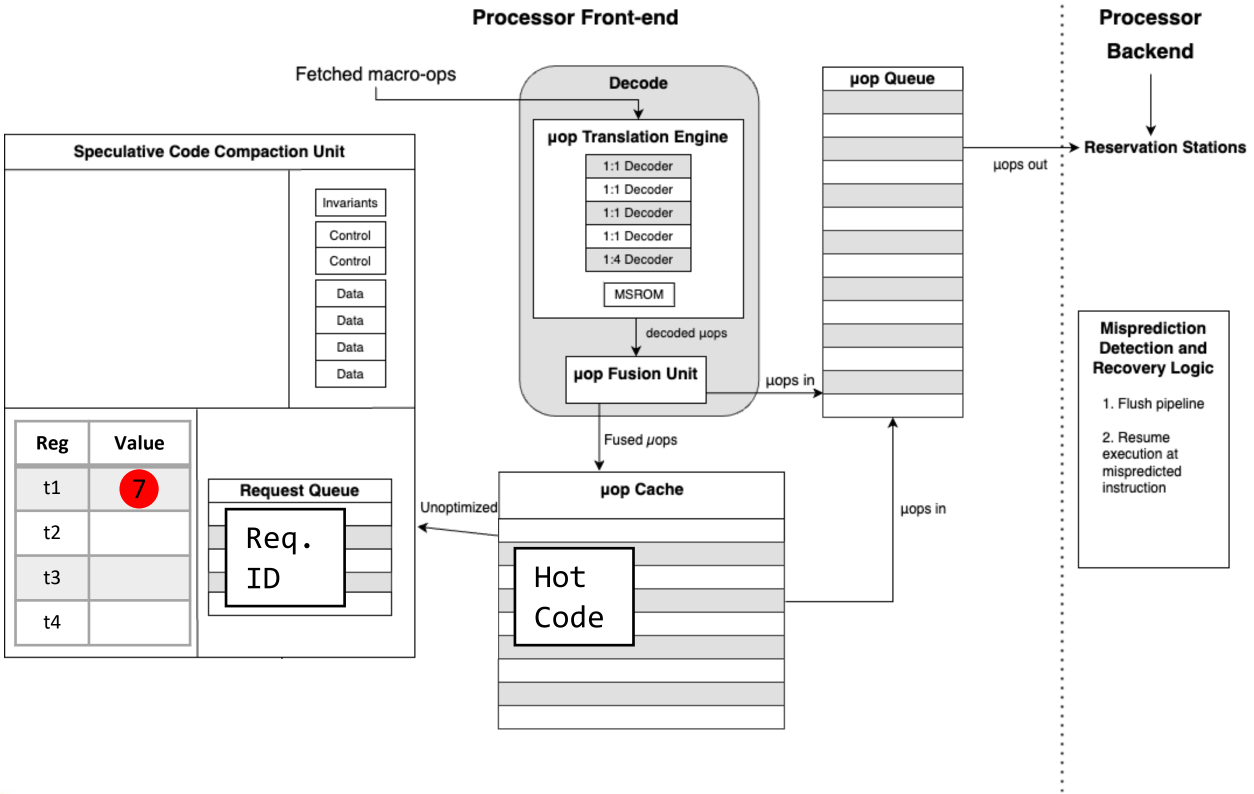


Step 3: Perform Optimizations

Speculative Data Invariant Identification – Value Prediction

```
     ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
      .
      .
      .
foo: add t4, t5, t6
```

Speculative Code Compaction

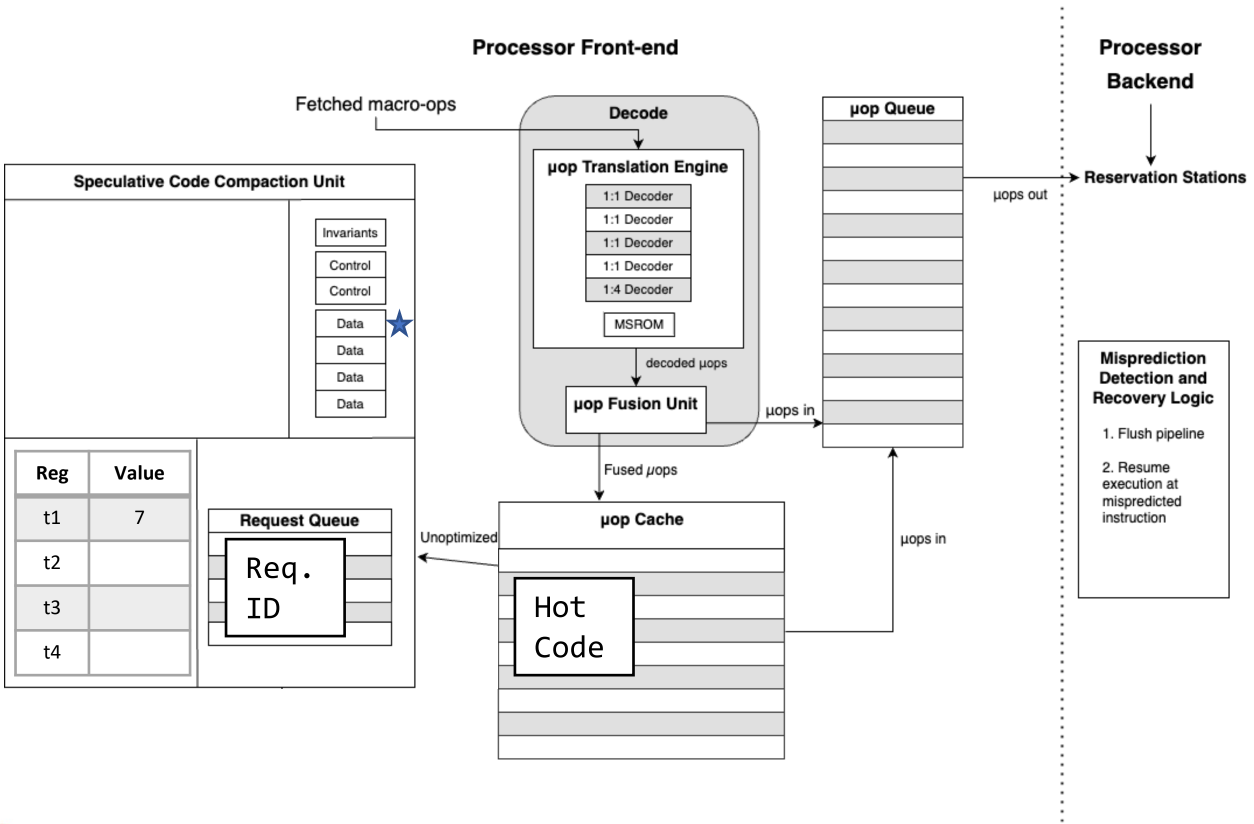


Step 3: Perform Optimizations

Speculative Data Invariant Identification – Value Prediction

```
➡ ★ld   t1, [ADDR]
     ld   t2, [ADDR + 8]
     addi t3, t2, 2
     beq  t1, t3, foo
       .
       .
       .
foo: add t4, t5, t6
```

Speculative Code Compaction

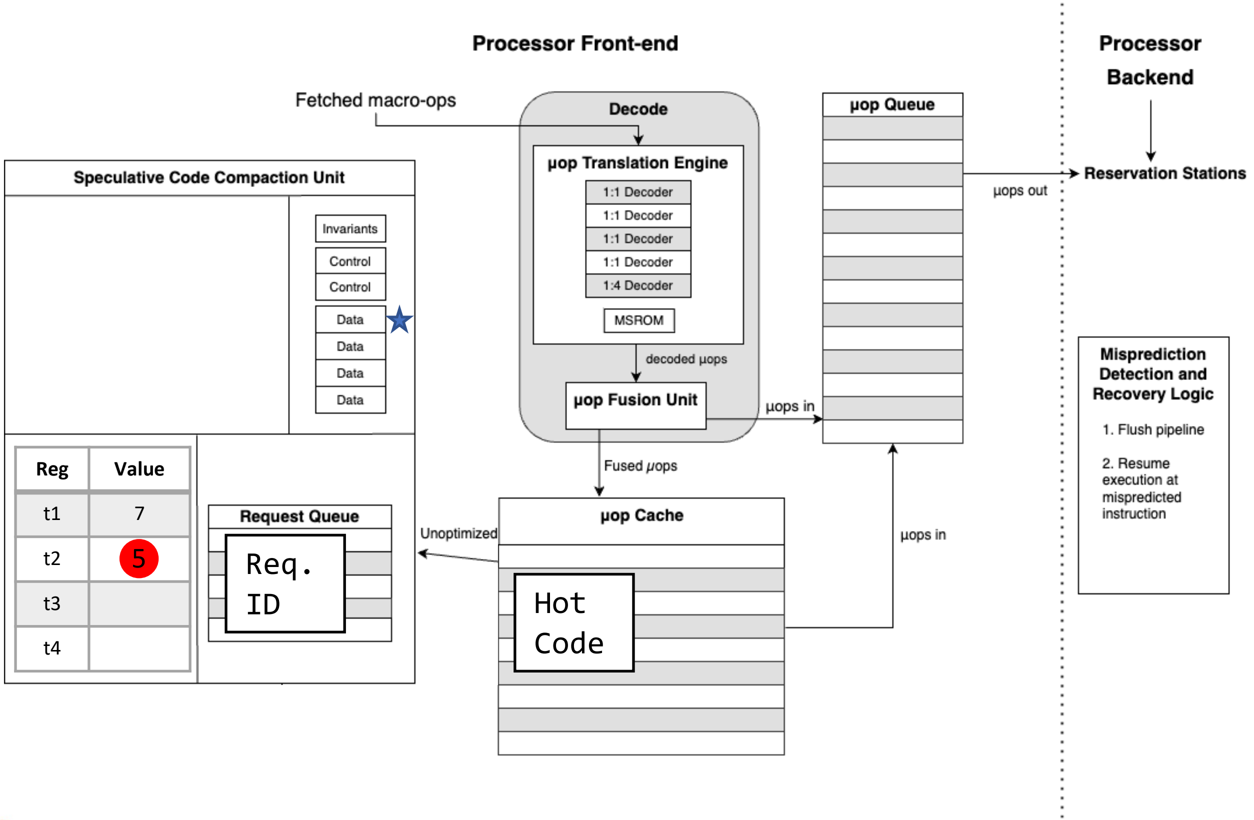


Step 3: Perform Optimizations

Speculative Data Invariant Identification – Value Prediction

```
  ★ld   t1, [ADDR]
➡ ld   t2, [ADDR + 8]
   addi t3, t2, 2
   beq  t1, t3, foo
     .
     .
     .
foo: add t4, t5, t6
```

Speculative Code Compaction

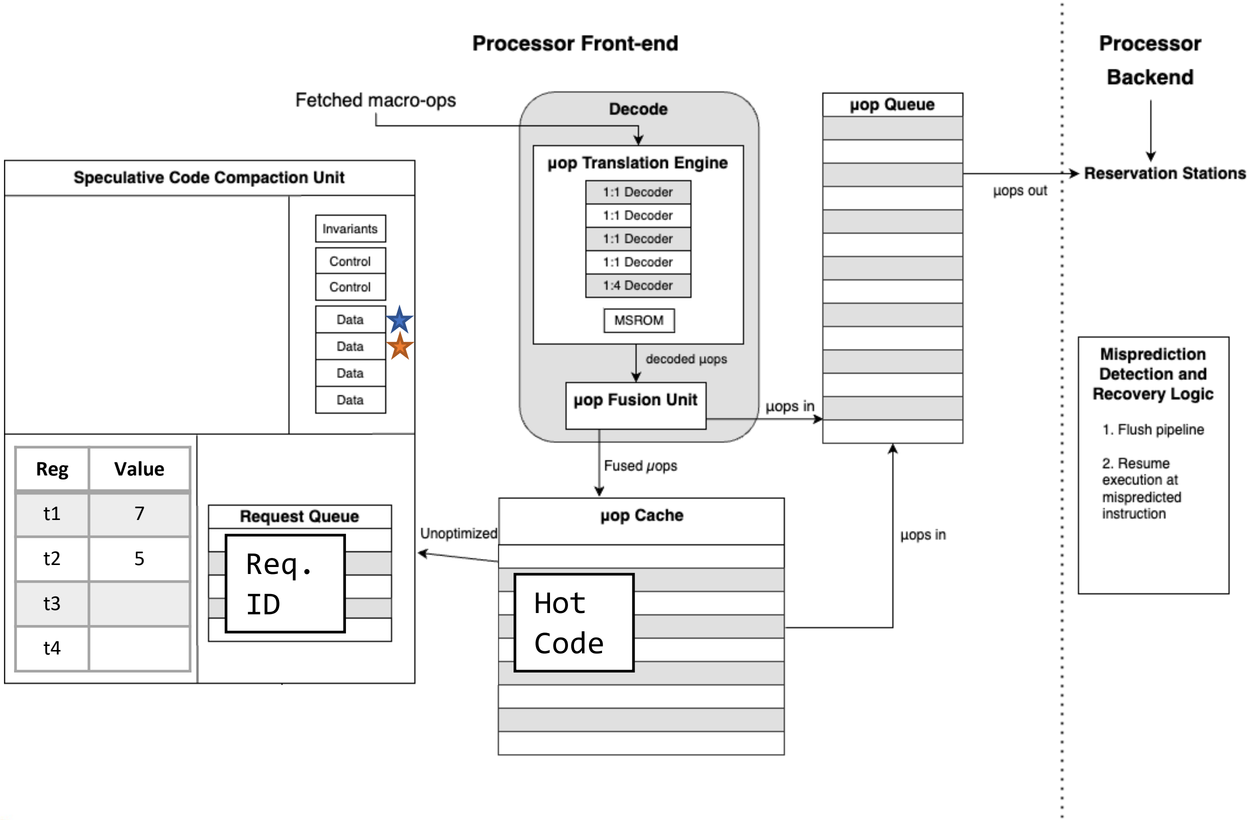


Step 3: Perform Optimizations

Speculative Data Invariant Identification – Value Prediction

```
★ld   t1, [ADDR]
★ld   t2, [ADDR + 8]
 addi t3, t2, 2
 beq  t1, t3, foo
   .
   .
   .
foo: add t4, t5, t6
```

Speculative Code Compaction

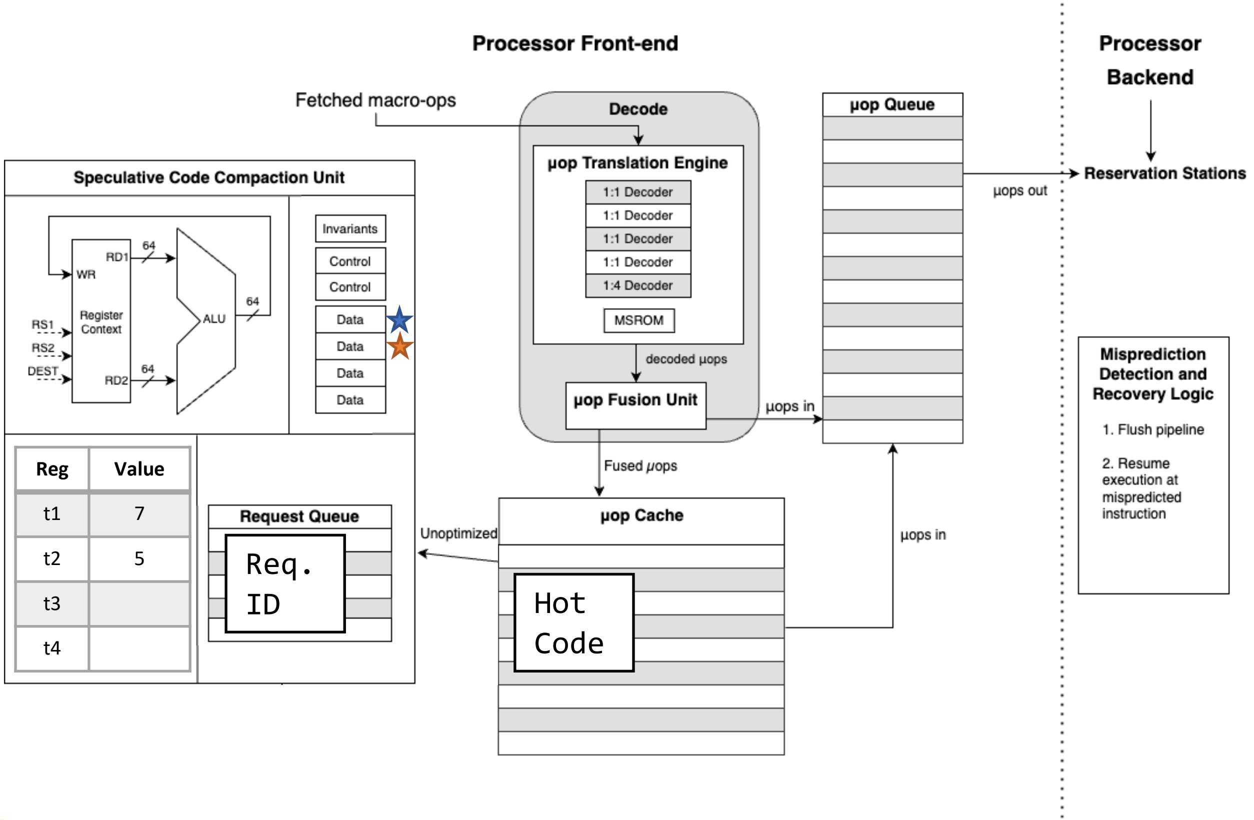


Step 3: Perform Optimizations

Constant Folding

```
  ld   t1, [ADDR]
  ld   t2, [ADDR + 8]
  addi t3, t2, 2
  beq  t1, t3, foo
   .
   .
   .
foo: add t4, t5, t6
```

Speculative Code Compaction

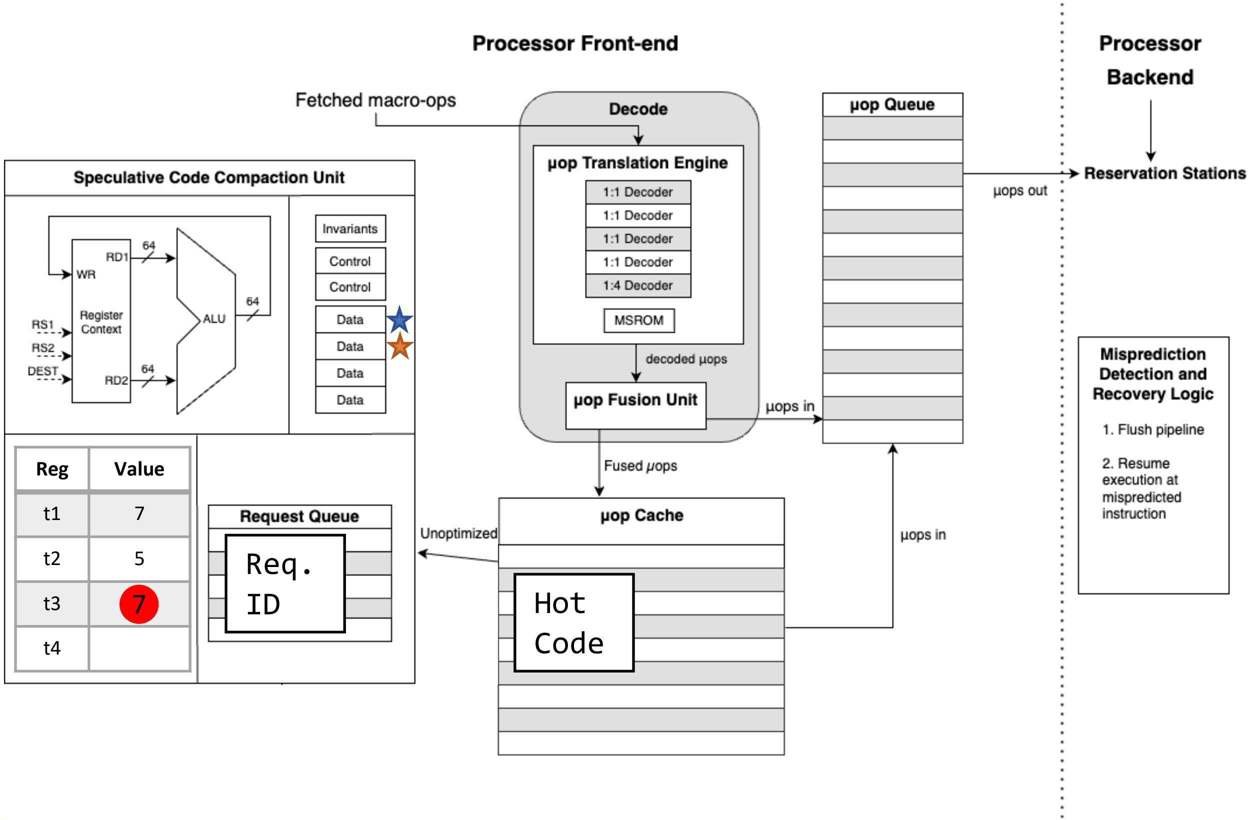


Step 3: Perform Optimizations

Constant Folding

```
★ld   t1, [ADDR]
★ld   t2, [ADDR + 8]
➜ addi t3, t2, 2
  beq  t1, t3, foo
   .
   .
   .
foo: add t4, t5, t6
```

Speculative Code Compaction

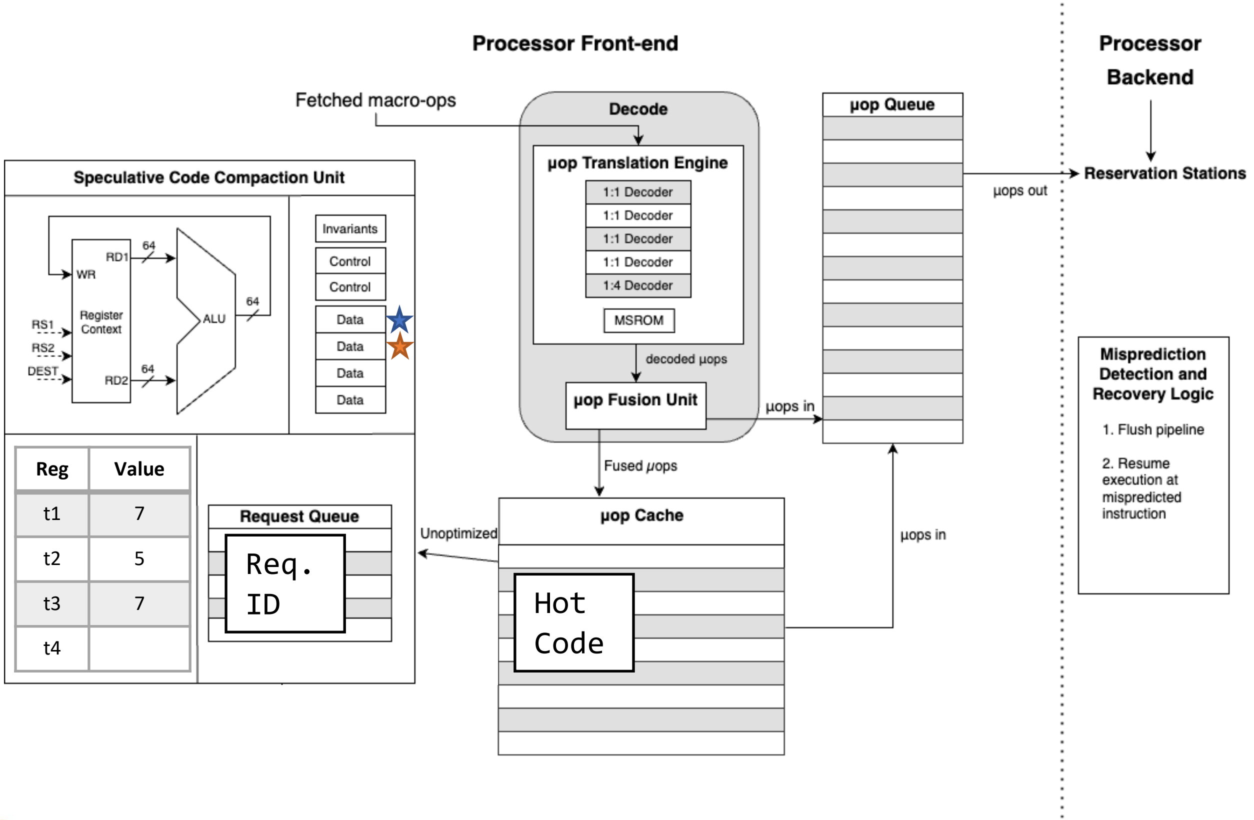


Step 3: Perform Optimizations

Constant Folding

```
      ld   t1, [ADDR]
      ld   t2, [ADDR + 8]
      addi t3, t2, 2
      beq  t1, t3, foo
        .
        .
        .
foo:  add t4, t5, t6
```

Speculative Code Compaction

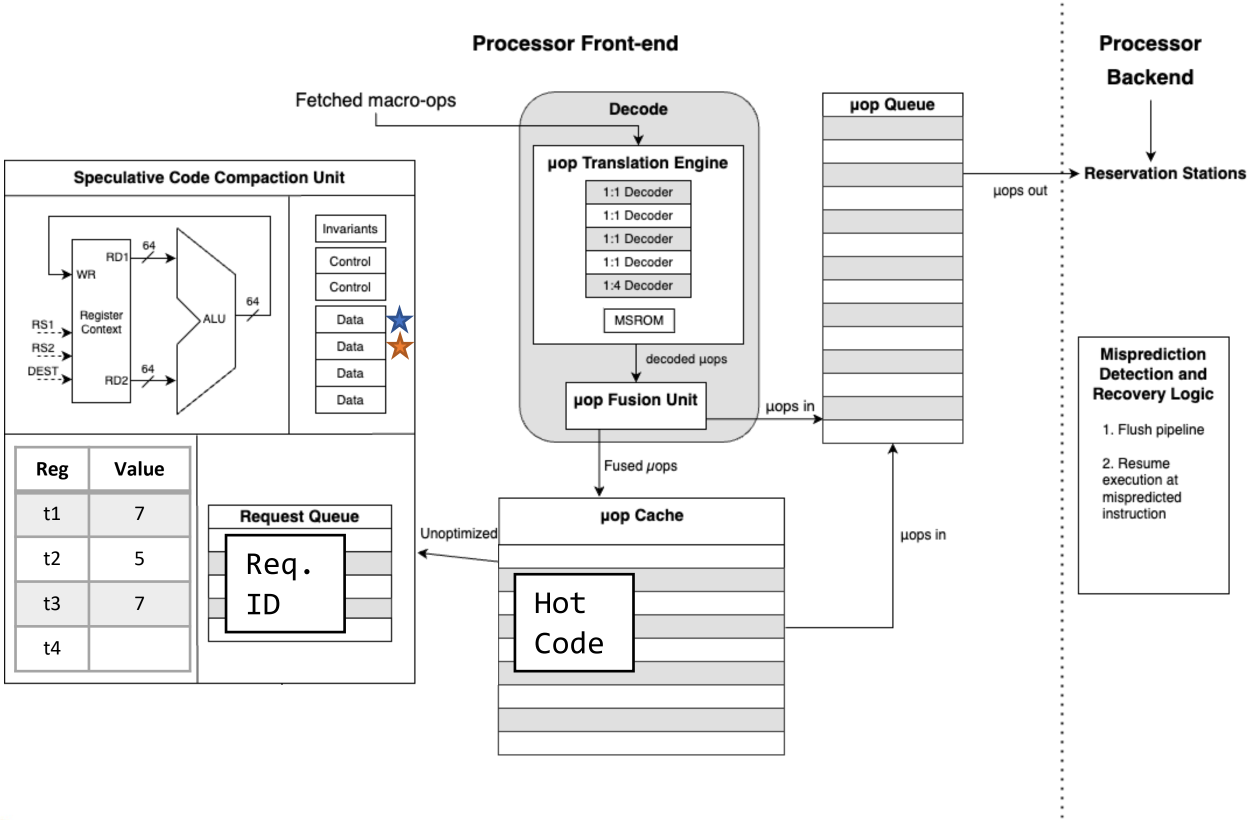


Step 3: Perform Optimizations

Dead code Elimination

```
    ld   t1, [ADDR]
    ld   t2, [ADDR + 8]
    ~~addi t3, t2, 2~~
    beq  t1, t3, foo
     .
     .
     .
foo: add t4, t5, t6
```

Speculative Code Compaction

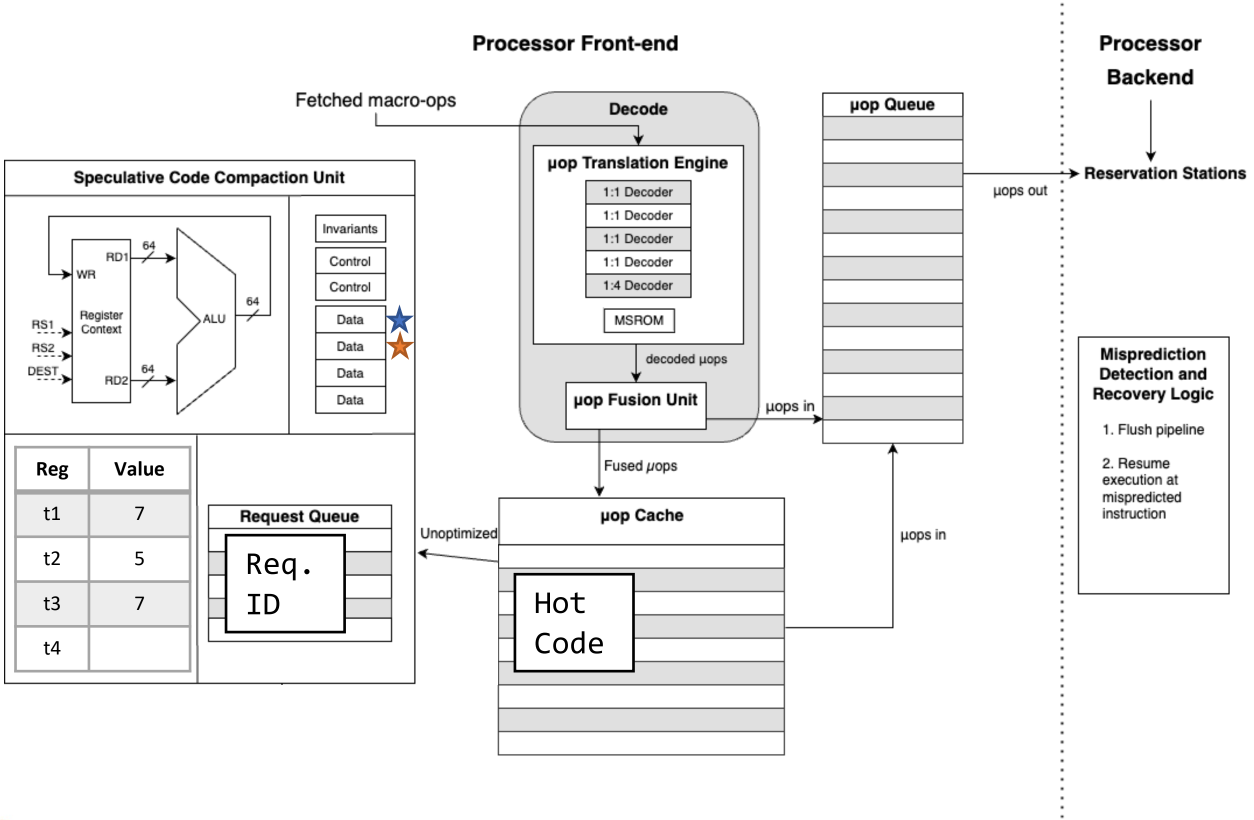


Step 3: Perform Optimizations

Constant Propogation

```
★ld   t1, [ADDR]
★ld   t2, [ADDR + 8]
 ~~addi t3, t2, 2~~
➡ beq  t1, t3, foo
  .
  .
  .
foo: add t4, t5, t6
```

Speculative Code Compaction

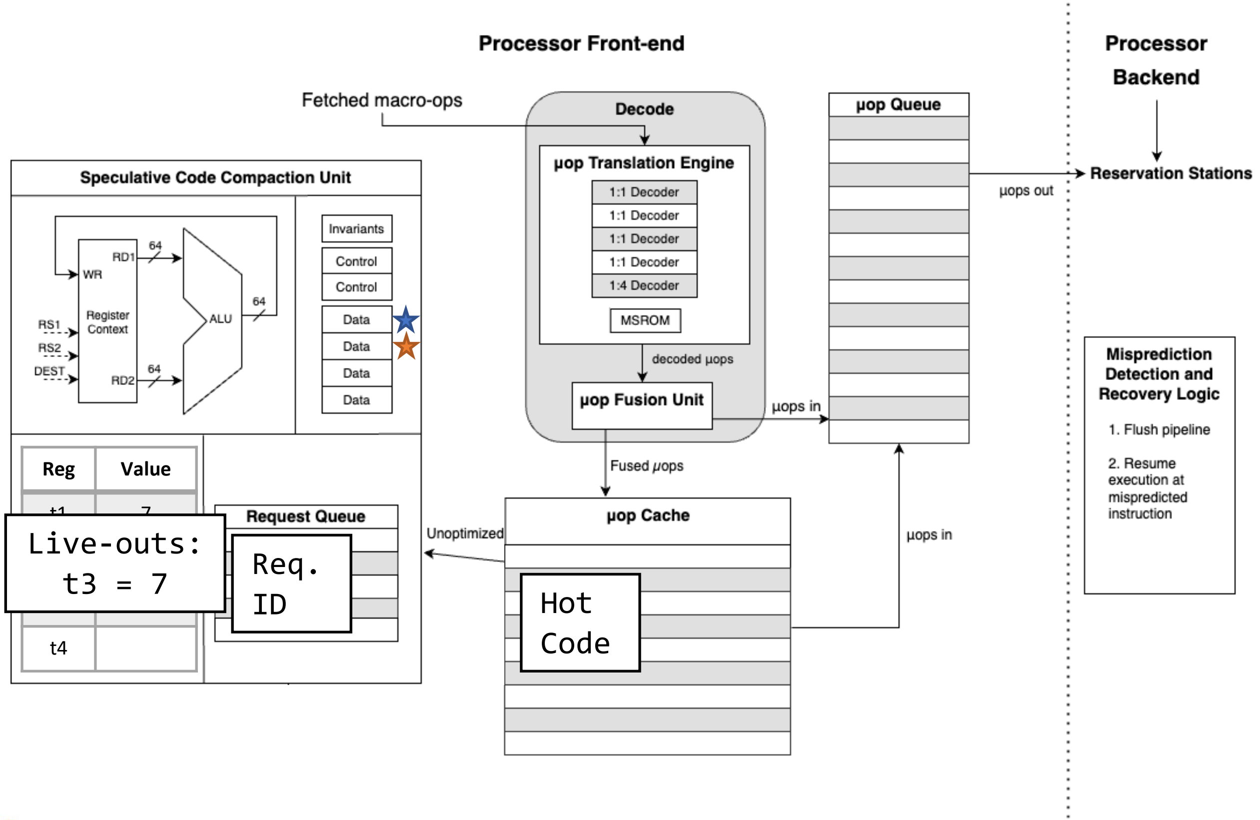


Step 3: Perform Optimizations

Branch Elimination

★ ld t1, [ADDR]
★ ld t2, [ADDR + 8]
~~addi t3, t2, 2~~
➡ ~~beq t1, t3, foo~~
add t4, t5, t6

Speculative Code Compaction

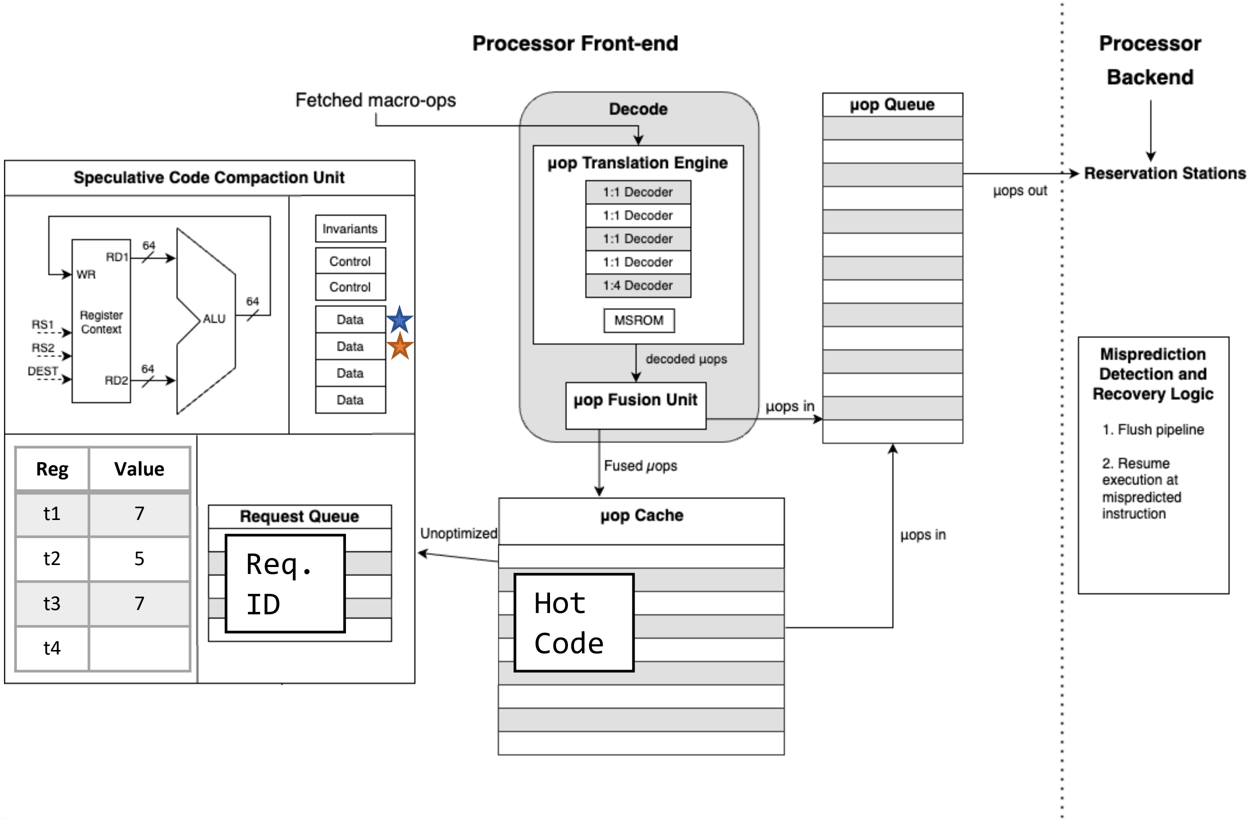


Step 4: Dump Live-outs

In order to maintain proper register state, we must dump live outs

```
★ld   t1, [ADDR]
★ld   t2, [ADDR + 8]
~~addi t3, t2, 2~~
~~beq  t1, t3, foo~~
add t4, t5, t6
```

Speculative Code Compaction



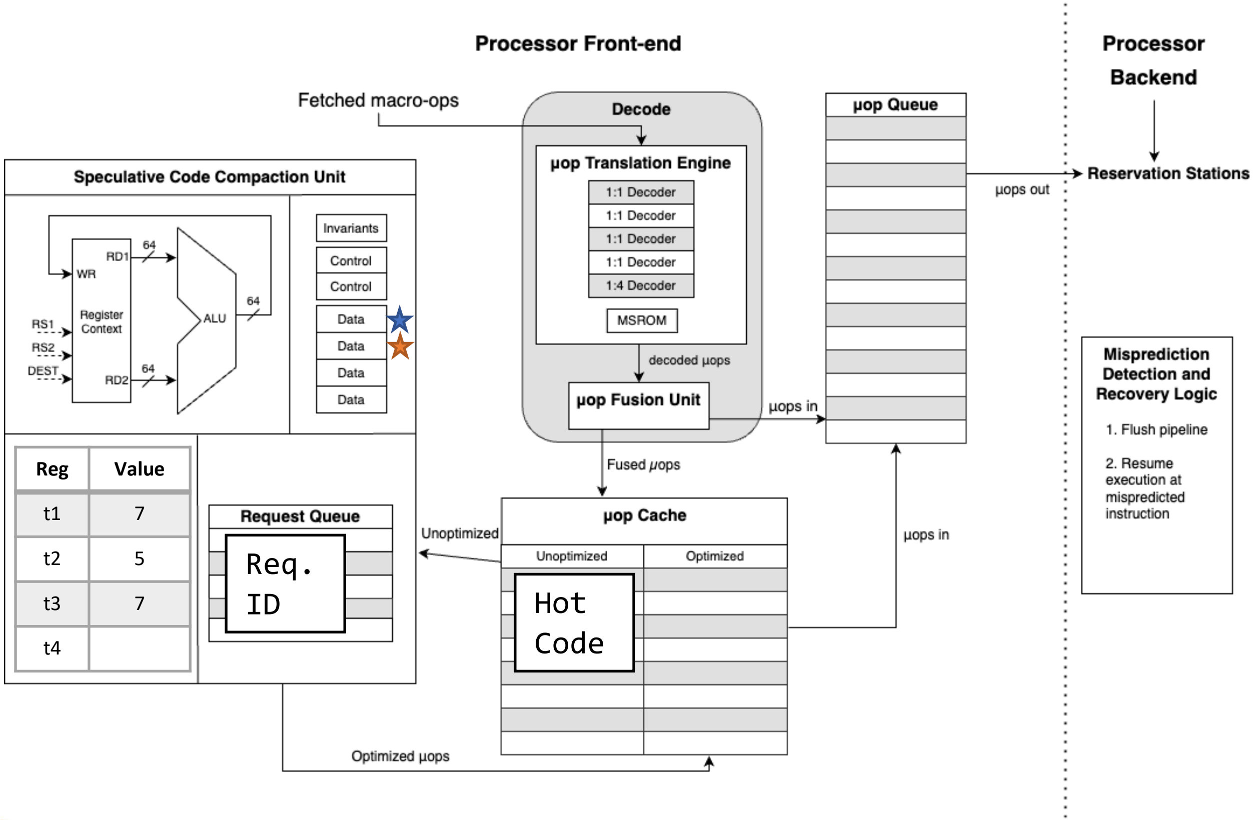
Step 4: Dump Live-outs

In order to maintain proper register state, we must dump live outs

★ld t1, [ADDR]
★ld t2, [ADDR + 8]
~~addi t3, t2, 2~~
~~beq t1, t3, foo~~
add t4, t5, t6

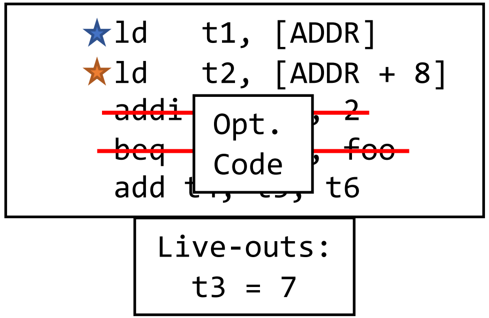
Live-outs:
t3 = 7

Speculative Code Compaction

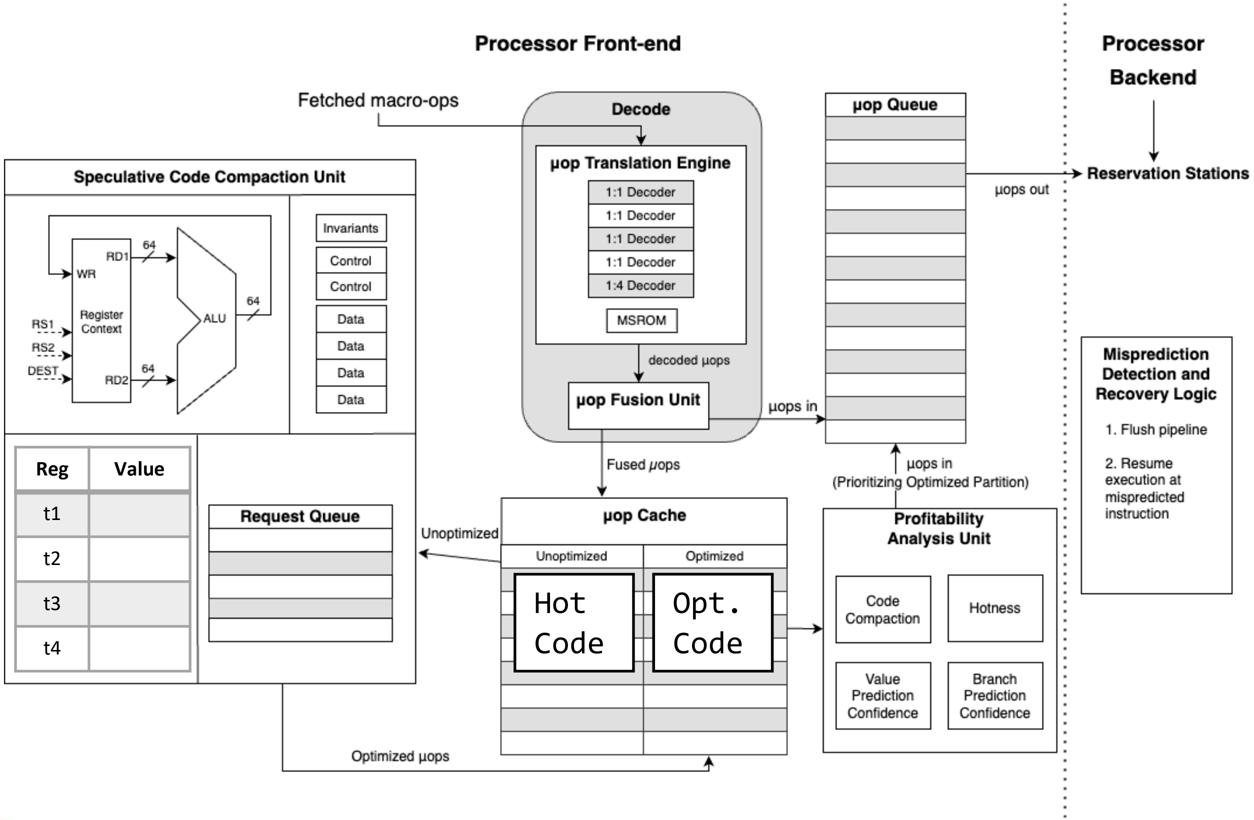


Step 5: Write to Optimized Partition

If there was sufficient shrinkage



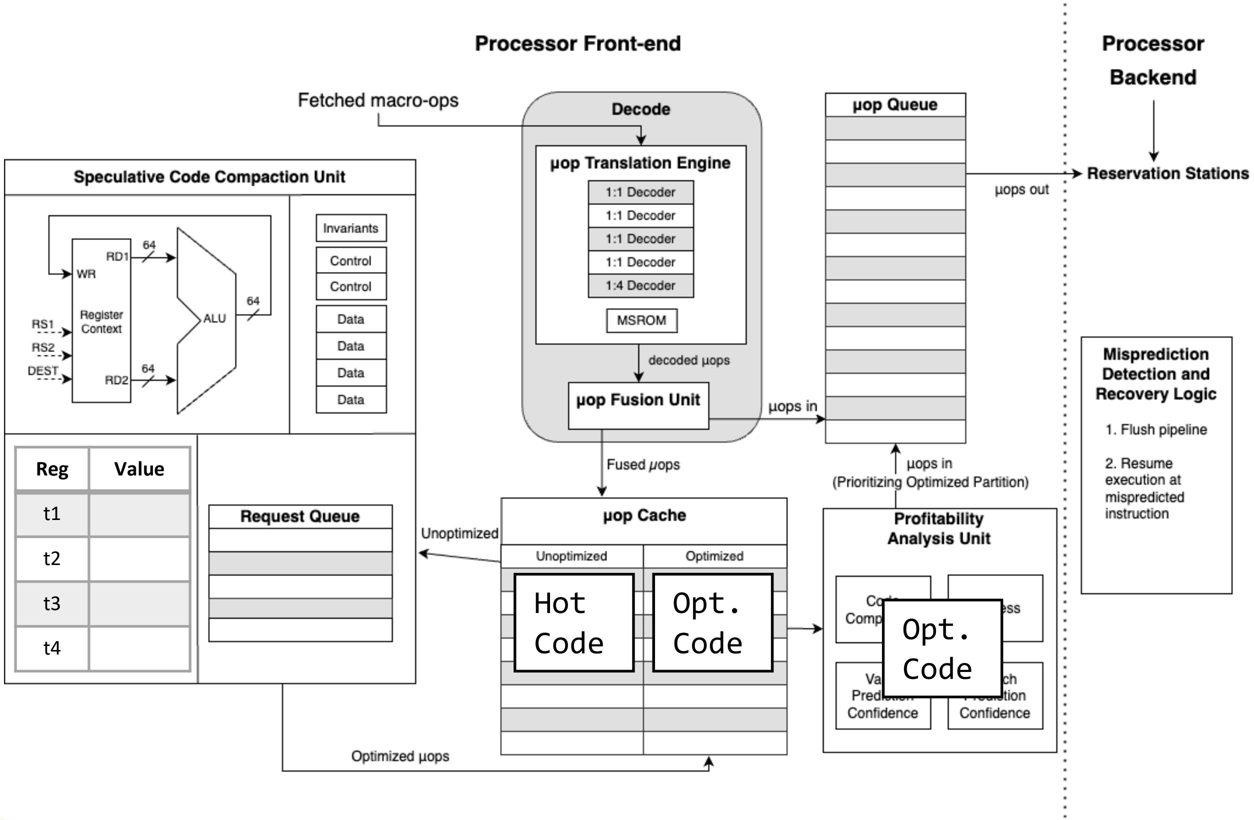
Speculative Code Compaction



Subsequent Executions

Next time the head PC is fetched, probe both partitions and perform profitability analysis

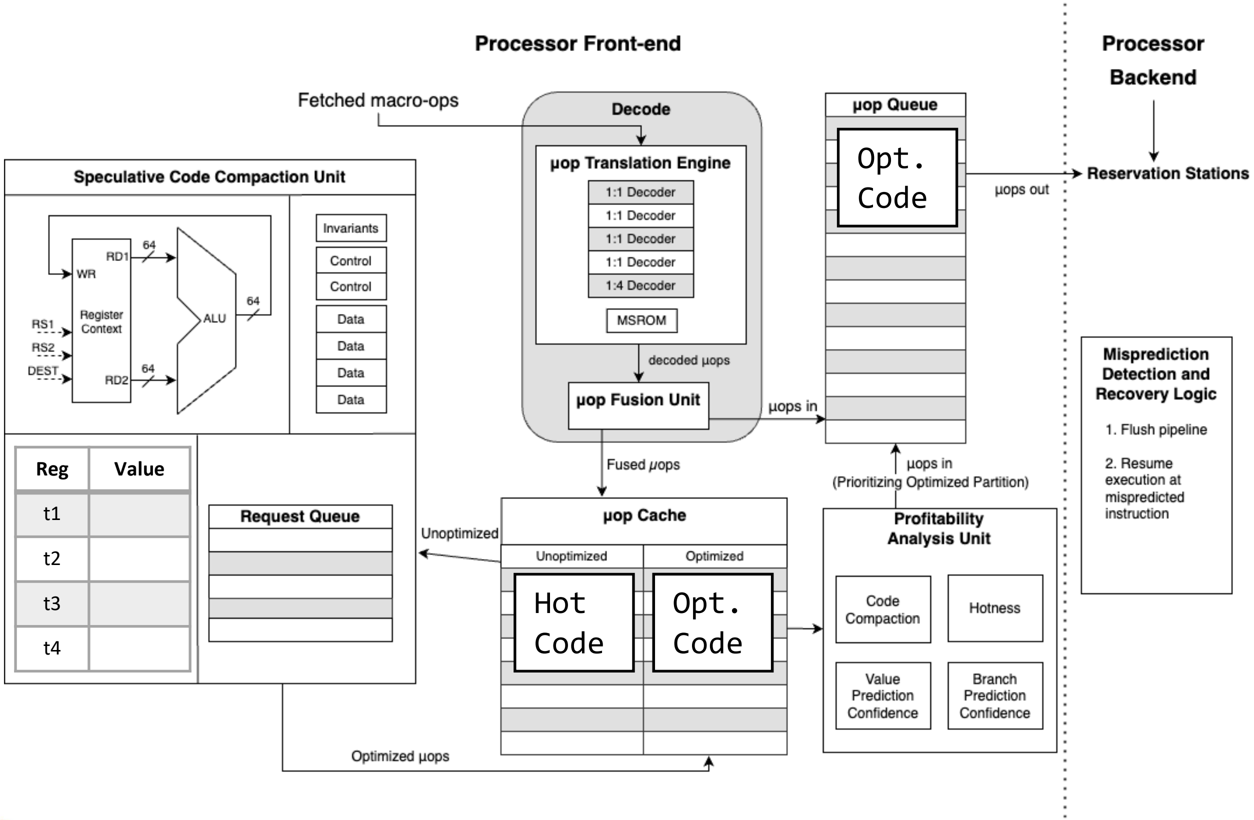
Speculative Code Compaction



Subsequent Executions

Next time the head PC is fetched, probe both partitions and perform profitability analysis

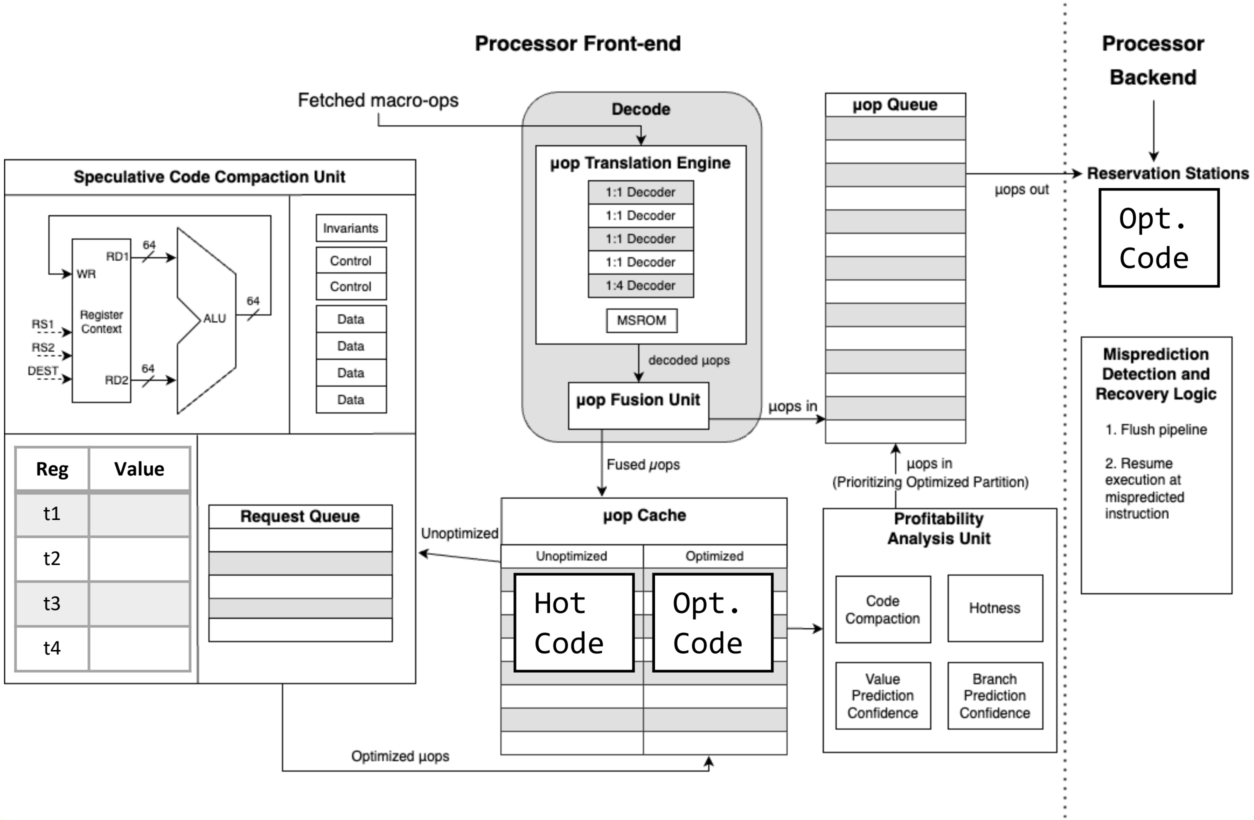
Speculative Code Compaction



Subsequent Executions

Next time the head PC is fetched, probe both partitions and perform profitability analysis

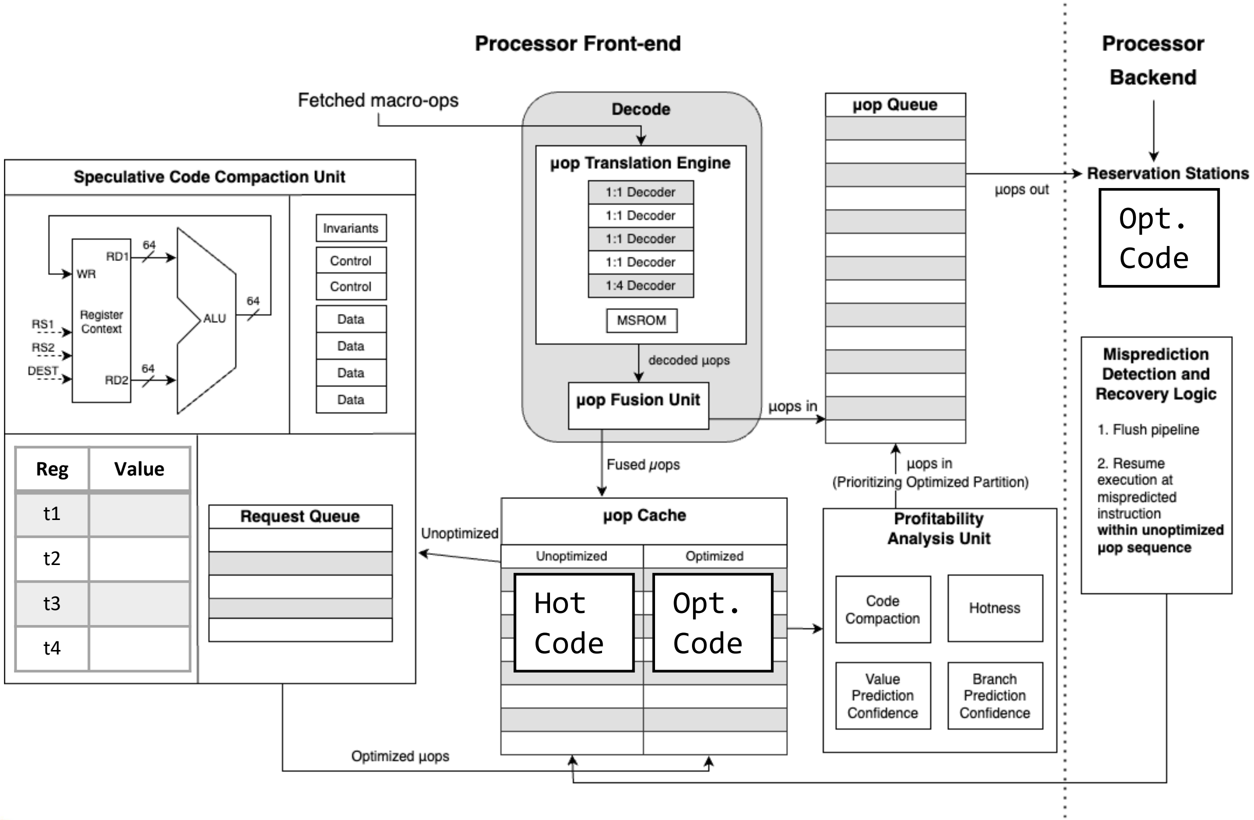
Speculative Code Compaction



Subsequent Executions

Next time the head PC is fetched, probe both partitions and perform profitability analysis

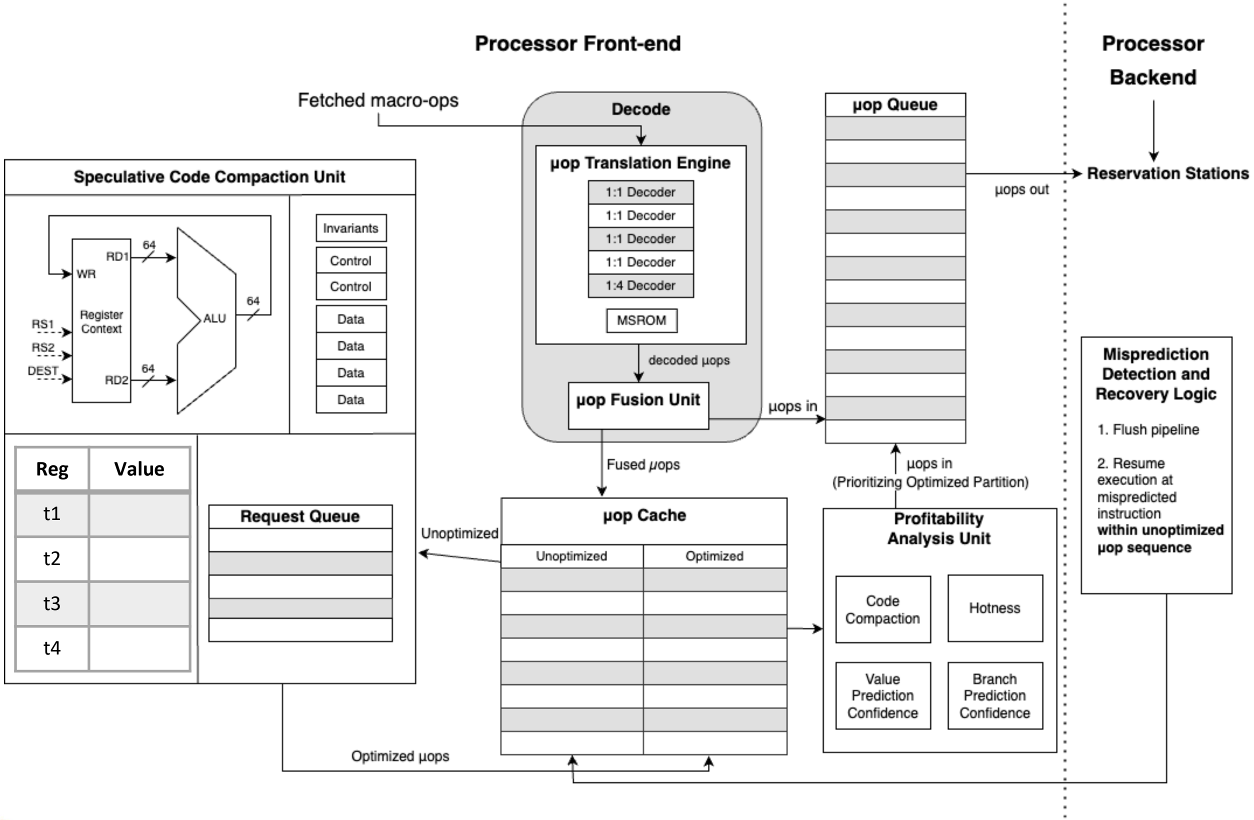
Speculative Code Compaction



Squashing and Recovery

If a prediction source is mispredicted, we must redirect execution to unoptimized sequence

Speculative Code Compaction



Optimizations:

- Data Invariant Identification
- Control Invariant Identification
- Constant Folding
- Constant Propagation
- Branch Folding
- Inlining Live Outs

Speculative Code Compaction

Motivation

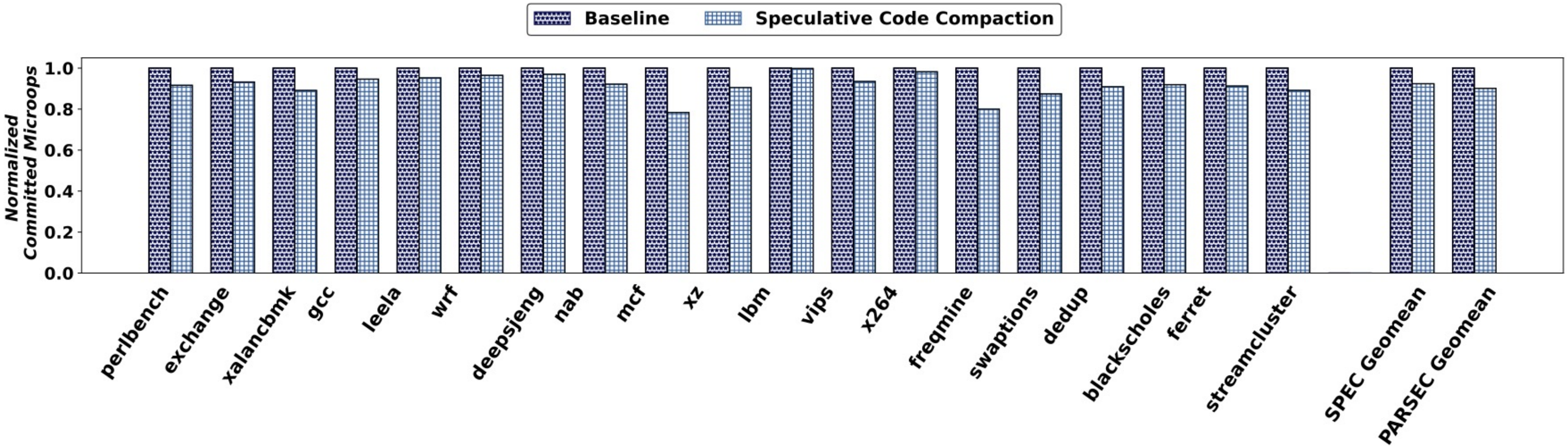
Overview of the Framework

Results

Conclusion

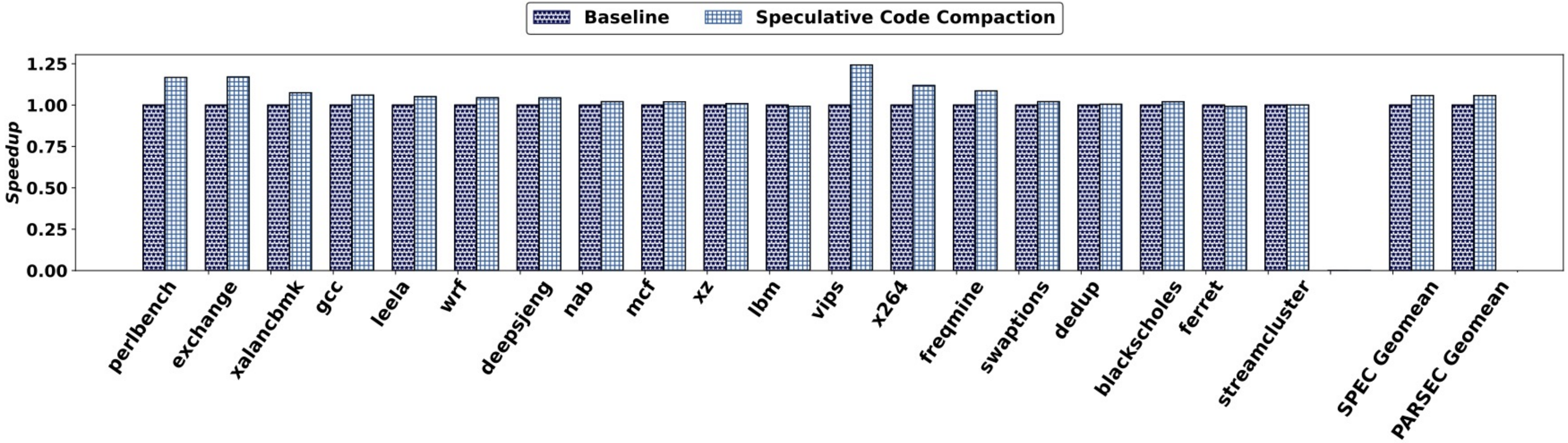
University of Virginia

Speculative Code Compaction



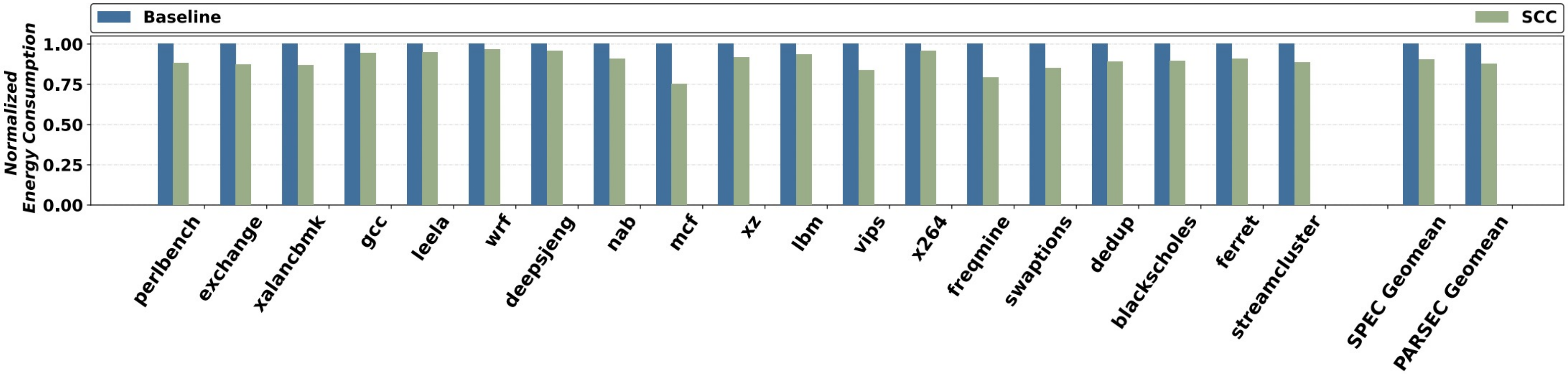
The majority of code compaction occurs within short, hot regions of code

Speculative Code Compaction



Benchmarks with high data and control predictability benefit the most from SCC

Speculative Code Compaction



SCC is able to reduce energy consumption even on applications which see no speedup

Speculative Code Compaction

- Motivation
- Overview of the Framework
- Results
- **Conclusion**

Speculative Code Compaction

- An aggressive scheme of dead code elimination implemented entirely within the processor front-end
- Minimally invasive (incurring just 1.5% in area overhead)
- Provides as much as 18% speedup (average of 6%) for SPEC applications
- Significant energy savings due to aggressive dead code elimination (an average of 12%)
- This research also involved several interesting explorations that study the sensitivity of our approach with different branch and value predictors
  - Aggressive prediction could lead to aggressive compaction, but also increases the risk of squashing, suggesting a balanced approach.

UNIVERSITY of VIRGINIA

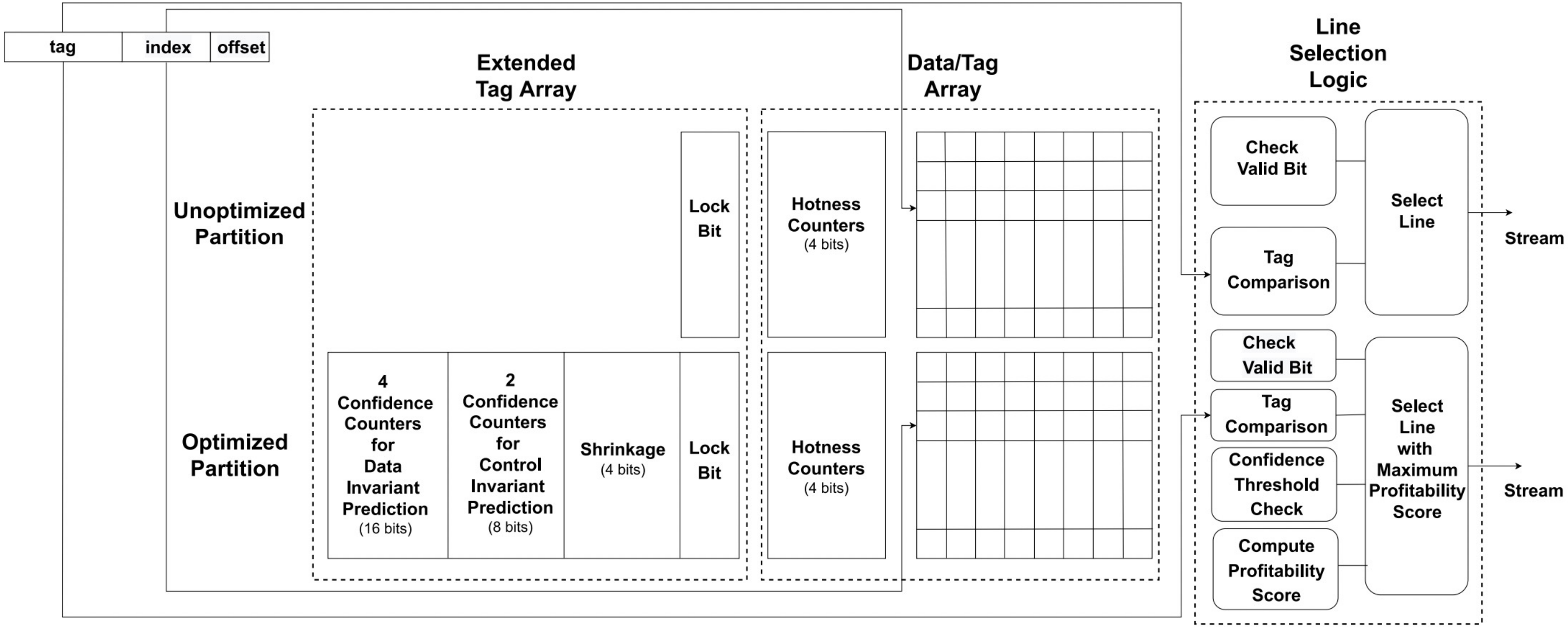
Thanks!

Questions?

www.github.com/logangregorym/gem5-changes

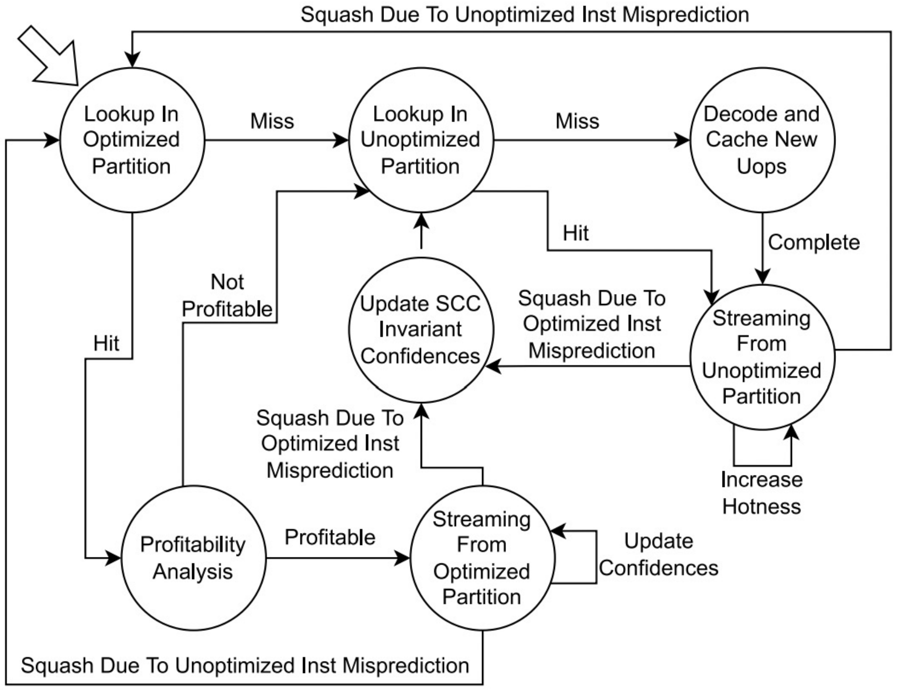
lgm4xn@virginia.edu

Extensions to The Micro-op Cache



Line selection logic extended to select line with highest profitability score

Fetch State Machine



Additional states and transitions added to handle streaming from optimized partition