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
Speculative Code Compaction

Motivation

Overview of the Framework

Results

Conclusion
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)
The Widening Hardware-Software Gap

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

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

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.

```c
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.

```java
int i = 0;
while (i < n) {
    int 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?
The Widening Hardware-Software Gap

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

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

Optimization 1

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

Optimization 2

```c
int 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

Processor Front-end
- Fetched macro-ops
- Decode
  - μop Translation Engine
    - 1:1 Decoder
    - 1:1 Decoder
    - 1:1 Decoder
    - 1:1 Decoder
    - 1:4 Decoder
    - MSROM
  - μop Fusion Unit
    - Fused μops
  - μop Cache
- μop Queue
  - μops in
  - μops out
- Reservation Stations

Processor Backend
- μop Queue

Intel Front-end
Legacy Decode and μop Cache

Misprediction Detection and Recovery Logic
1. Flush pipeline
2. Resume execution at mispredicted instruction
Speculative Code Compaction

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 1: Hot Code Detection
Identify regions of hot code in μop cache

```
1d t1, [ADDR]
1d 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

```
1d  t1, [ADDR]
1d  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 \( \mu \)op 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

```
1d  t1, [ADDR]
1d  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

```
1d t1, [ADDR]
1d 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

```
1d t1, [ADDR]
1d 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

\[
\begin{align*}
\text{ld} & \quad t1, [ADDR] \\
\text{ld} & \quad t2, [ADDR + 8] \\
\text{addi} & \quad t3, t2, 2 \\
\text{beq} & \quad t1, t3, foo \\
\text{add} & \quad t4, t5, t6
\end{align*}
\]
Speculative Code Compaction

Step 4: Dump Live-outs
In order to maintain proper register state, we must dump live outs

```
1d  t1, [ADDR]
1d  t2, [ADDR + 8]
add 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.

```
    1d  t1, [ADDR]
    1d  t2, [ADDR + 8]
    addi t3, t2, 2
    beq t1, t3, foo
    addi 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.
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

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

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

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.
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
Additional states and transitions added to handle streaming from optimized partition