SPUR

A TRACE-BASED JIT COMPILER FOR .NET
(MAKES JAVASCRIPT SUPER FAST)

Nikolai Tillmann
Microsoft Research

http://research.microsoft.com/Spur/

VMIL at SPLASH, Reno, October 2010

Introduction

- Spur: One runtime + JIT for all languages on .NET!
- Tracing JIT, advanced optimizations
- Platform for research experiments

Background: JavaScript

```
var sum = 0
for (var i = 0; i < 1000; i++) {
     if (i == 990) {
          sum += " Hello World "
     }
     sum += 1
}

print(sum)
```

result: "990 Hello World 1111111111"

- Dynamically typed, type inference is difficult in the general case
- The semantics of "add" depend on its operands, this results in lots of runtime checks

Basic idea: Trace Compilation

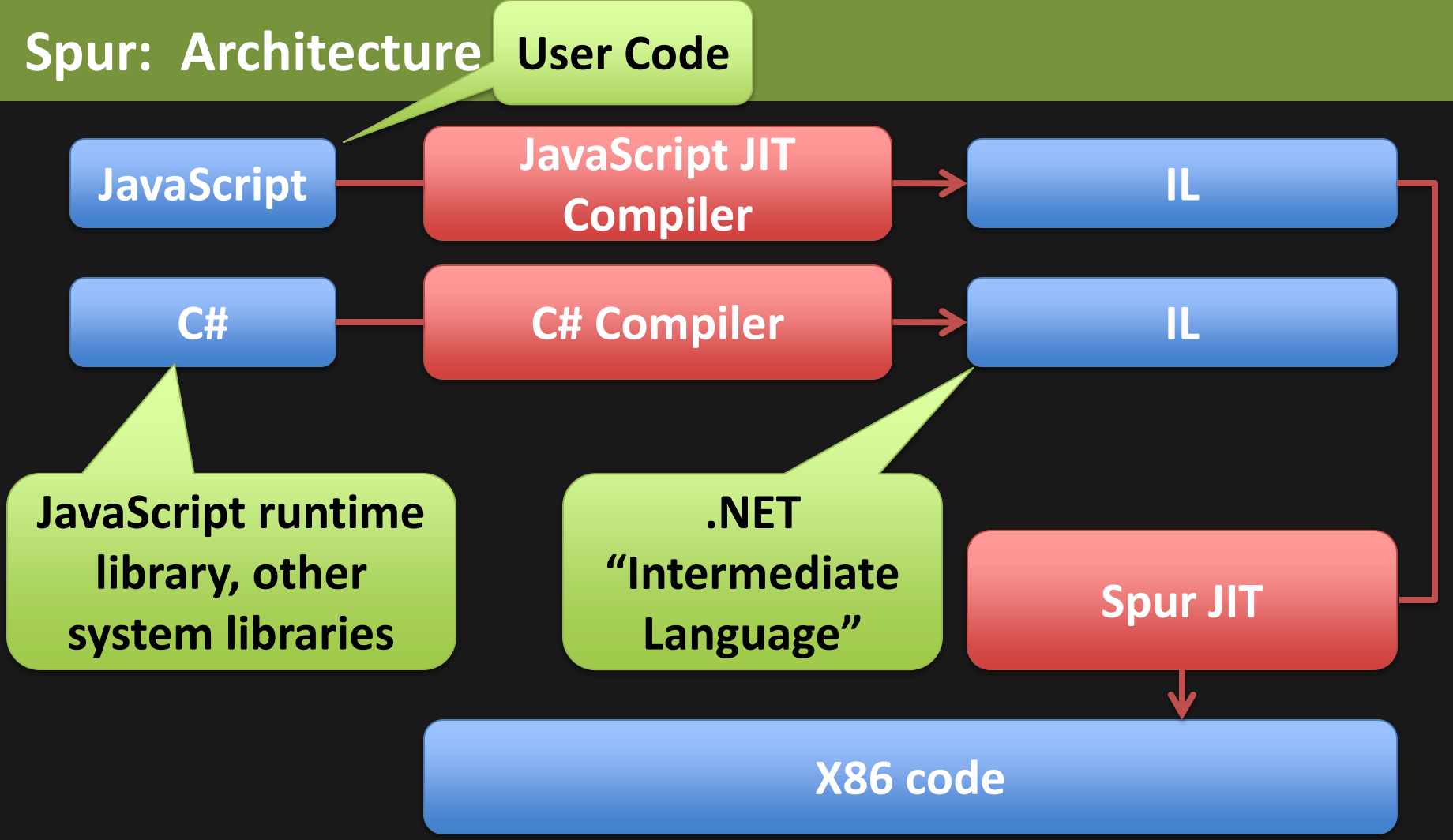
```
var sum = 0
for (var i = 0; i < 1000; i++) {
     if (i == 990) {
          sum += " Hello World "
     }
     sum += 1
}

print(sum)
```

result: "990 Hello World 1111111111"

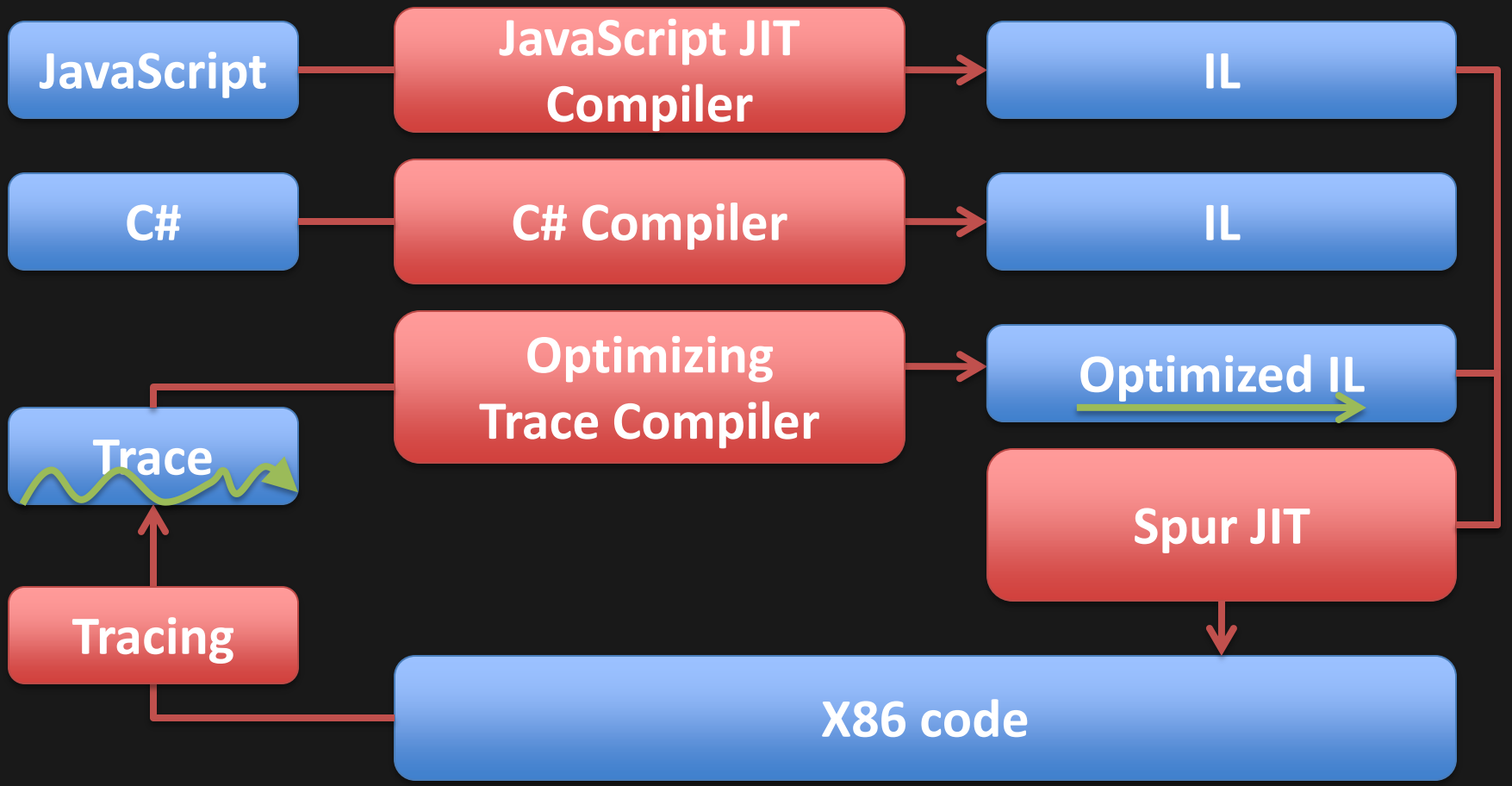
- Trace compilation of loop for case where "sum" is an integer
- After "sum" becomes a string,
  execution is resumed in un-optimized code

Spur: Architecture



- Java Script source compiled to IL, references JavaScript Runtime written in C#
- Compiled Script and JavaScript Runtime are JITted to x86 code

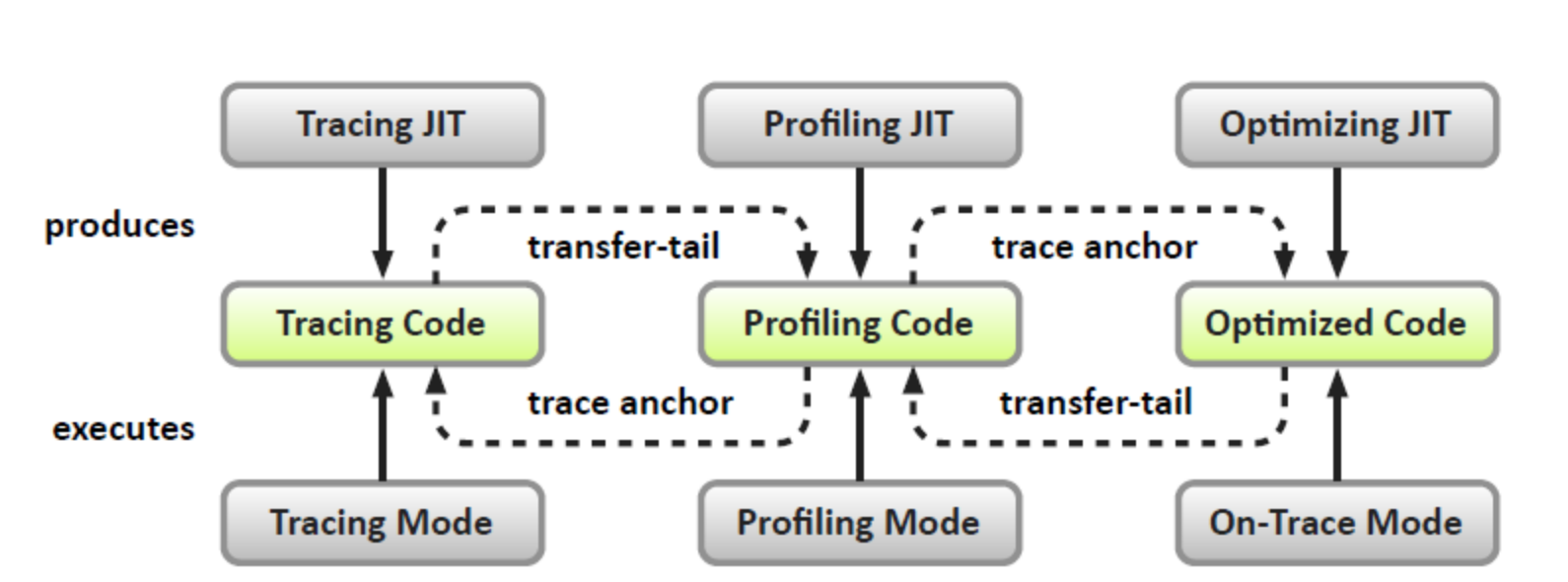
Spur: Architecture



Spur traces and optimizes the combination of

- the JavaScript IL Code and the
- JavaScript Runtime IL Code

Runtime: Code Version



Spur initially generates Tracing Code:

simple JIT with loop counters, only block-level register allocation

If loop is hot, it generates Profiling Code:

same as tracing code, but with call backs to monitor trace

If execution stays on trace, it generates Optimized Code

implements all standard (and some speculative) optimizations

Which means, that we go from this ...

```
for (var n = 0; n < 1000; n++) {
    for (var n2 = 0; n2 < 1000; n2++) {
        for (var i = 0; i < a.length - 1; i++) {
            var tmp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = tmp;
        }
    }
}
```

- 35 method calls, 129 guards, 224 total instructions

to this ...

back to IL code equivalent to this C# code

```
double index = 0;

/* ... lots of hoisted code */
while (true) {
    if (index >= arrayIndex - 1) {
        // transfer
    }

    if (index + 1 >= validArrayIndex) {
        // transfer
    }

    /* bounds checks have been proven */
    var temp = array[(int) index];
    array[(int)index] = array[(int)index + 1];
    array[(int) index + 1] = temp;

    index += 1;
}
```

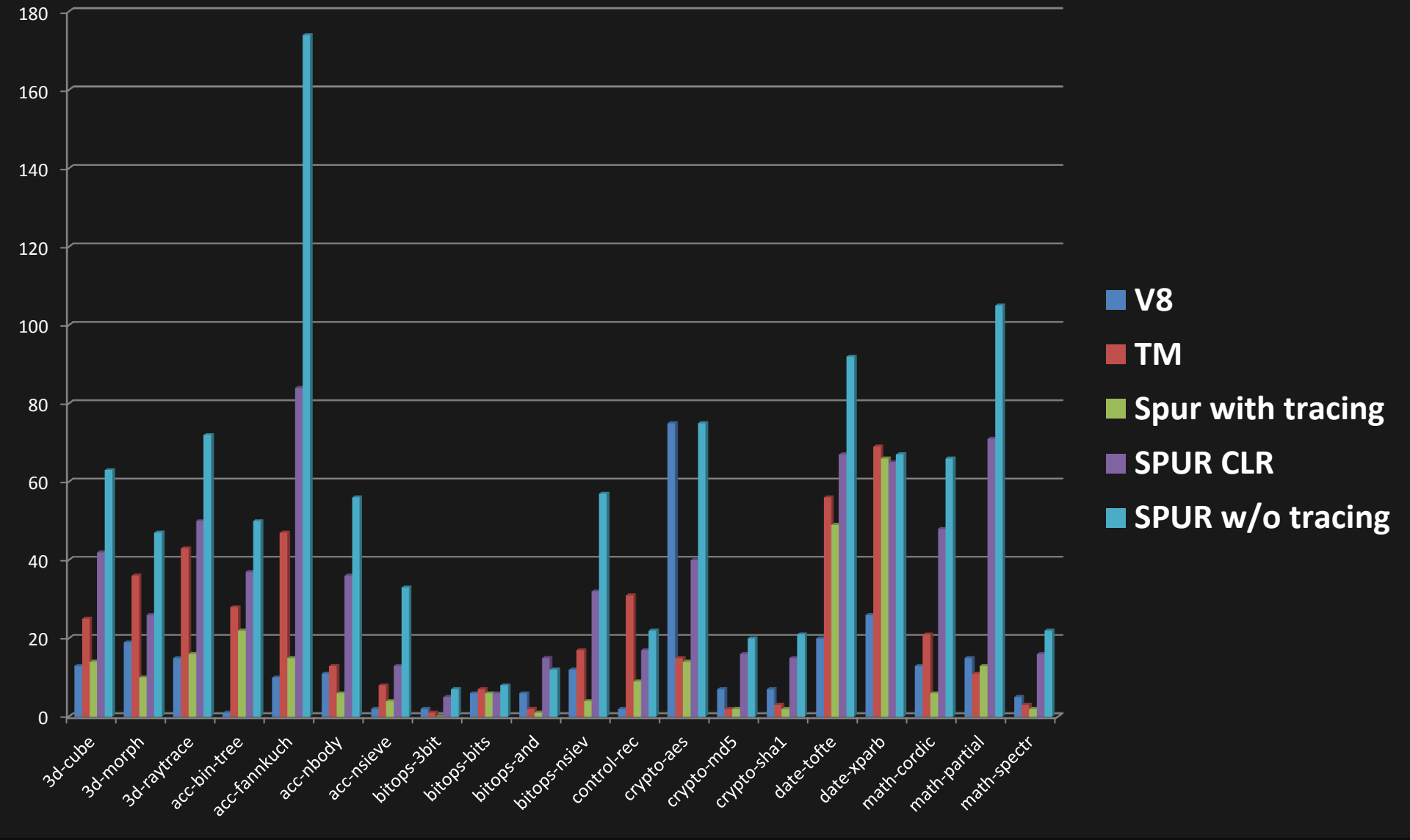
- 41 loop prologue instructions, 18 loop prologue guards
- 10 loop instructions, 2 loop guards!
- From 224 to 10, performance is comparable the C# version of the loop, 7x faster than the CLR operating on JScript code, and slightly faster than V8

Optimizing Trace Compiler

Standard and not-so-standard optimizations:

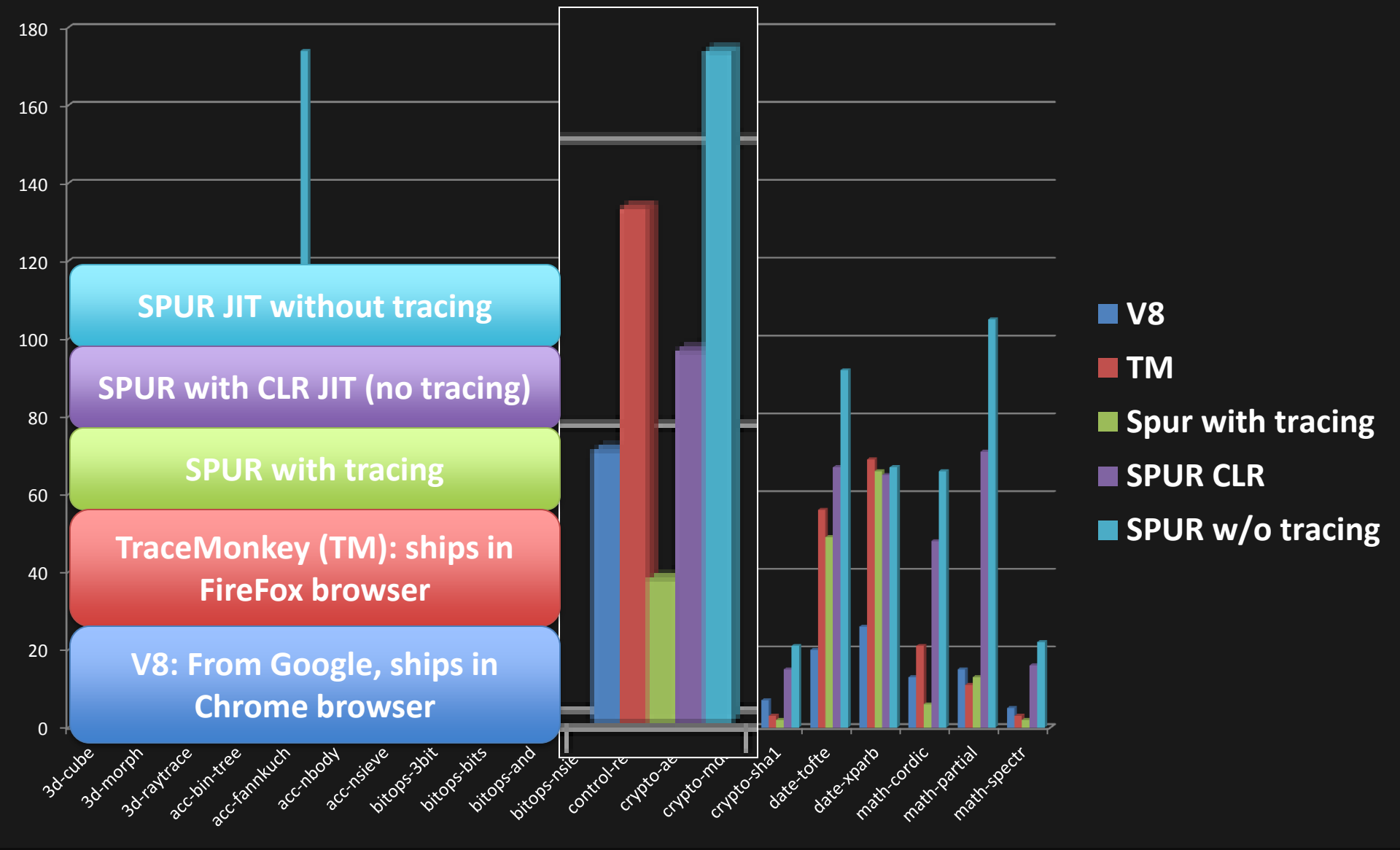
- Inlining (for free via tracing)
- Loop unfolding
- Dead Code Eliminations
- Invariant Code Motion
- Constant Folding
- Expression simplification
- Common Subexpression Elimination
- Alias Analysis
- Redundant Guard Elimination
- Redundant Load/Store Elimination
- Speculative Guard Strengthening

Numbers (SunSpider JavaScript benchmarks)



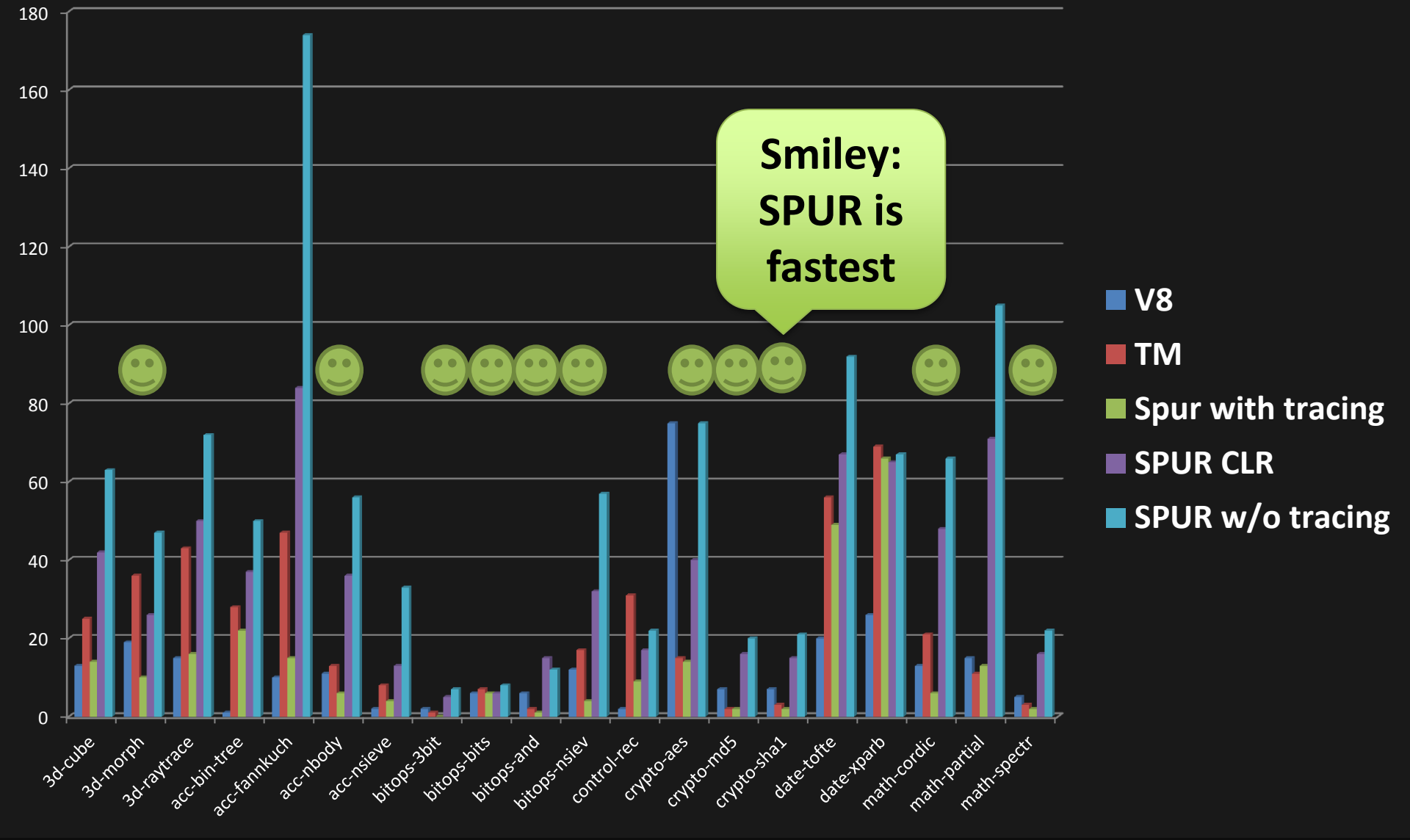
- Execution time in milliseconds. Smaller is better.

Numbers (SunSpider JavaScript benchmarks)



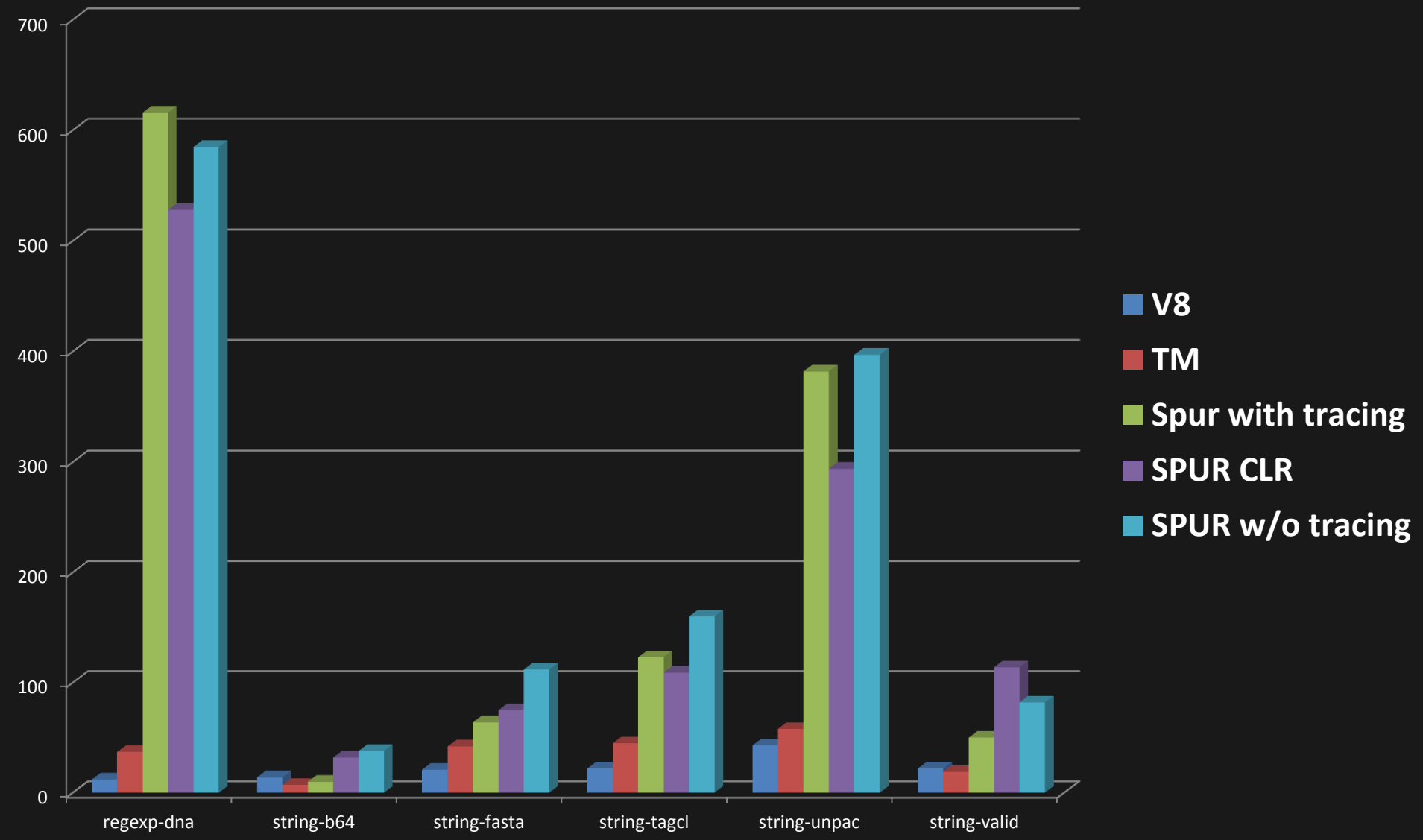
- Execution time in milliseconds. Smaller is better.

Numbers (SunSpider JavaScript benchmarks)



- Execution time in milliseconds. Smaller is better.

Numbers (SunSpider benchmarks cont.)



- Execution time in milliseconds. Smaller is better.

SPUR

A WALKTHROUGH: JAVASCRIPT

Basic idea: Trace Compilation

```
var sum = 0
for (var i = 0; i < 1000; i++) {
     if (i == 990) {
          sum += " Hello World "
     }
     sum += 1
}

print(sum)
```

- How would this read in C#?

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2, i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind
  double DoubleV
  object ObjectV
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

- When a guard fails, we jump back to unoptimized code
- When a guard fails often, another trace would be recorded

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2, i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Guard strengthening

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2, i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Inlining

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2, i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Invariant Code Motion

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2, i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Dead-code elimination

Raw recorded trace:

```
// initial state in trace recording:
//   sum.Kind == Double
//   sum.DoubleValue == 3
//   i == 3
begin:
  guard i < 1000
  guard i != 990
  begin AddDouble
    guard sum.Kind == Double
    t1 := sum.DoubleValue + 1.0D
    t2 := Value { Kind = Double;
                  DoubleValue = t1; }
  end AddDouble
  t3 = i + 1
  // update state, loop back
  sum := t2; i := t3
  goto begin
```

- SSA form
- At IL level

Walkthrough

C# equivalent code:

```
enum ValueKind { Double, Object, … }
struct Value {
  ValueKind Kind;
  double DoubleValue;
  object ObjectValue;
  …
}

void Main() {
  Value sum;
  SetToDouble(ref sum, 0.0D);
  for (int i = 0; i < 1000; i++) {
    if (i == 990) {
      AddObject(ref sum, " Hello World “);
    }
    AddDouble(ref sum, 1.0D);
  }
  Console.WriteLine(ToString(sum));
}
```

Optimized trace:

```
prologue:
  guard i < 990
  guard sum.Kind == Double

loopBegin:
  guard i != 990
  t1 := sum.DoubleValue + 1.0D
  // update state, loop back
  i = i + 1
  (&sum)->DoubleValue = t1
  goto loopBegin
```

SPUR

ANOTHER EXAMPLE: C# (NO JAVASCRIPT)

Fluffy C#

```
class Container {
  private Data data;
  public virtual Data Data {
    get {
      if (data == null) data = new Data();
      return data;
    }
  }
}
class Data {
  private int[] elements = new int[1000];
  public virtual int Count {
    get { return elements.Length; } }
  public virtual int GetElement(int index)
  {
    if (index < 0 || index > elements.Length)
      throw new ArgumentOutOfRangeException();
    return elements[index];
  }
}
```

```
int Find(Container c,
         int element)
{
  for (int i = 0;
       i < c.Data.Count;
       i++)
  {
    if (c.Data.GetElement(i) ==
        element)
      return i;
  }
  return -1;
}
```

Fluffy C#

```
class Container {
  private Data data;
  public virtual Data Data {
    get {
      if (data == null) data = new Data();
      return data;
    }
  }
}
class Data {
  private int[] elements = new int[1000];
  public virtual int Count {
    get { return elements.Length; } }
  public virtual int GetElement(int index)
  {
    if (index < 0 || index > elements.Length)
      throw new ArgumentOutOfRangeException();
    return elements[index];
  }
}
```

```
int Find(Container c,
         int element)
{
  for (int i = 0;
       i < c.Data.Count;
       i++)
  {
    if (c.Data.GetElement(i) ==
        element)
      return i;
  }
  return -1;
}
```

- Virtual calls, lazy initialization, (repeated) argument validation, …

Fluffy C#: CLR vs. Spur

```
class Container {
  private Data data;
  public virtual Data Data {
    get {
      if (data == null) data = new Data();
      return data;
    }
  }
}
class Data {
  private int[] elements = new int[1000];
  public virtual int Count {
    get { return elements.Length; } }
  public virtual int GetElement(int index)
  {
    if (index < 0 || index > elements.Length)
      throw new ArgumentOutOfRangeException();
    return elements[index];
  }
```

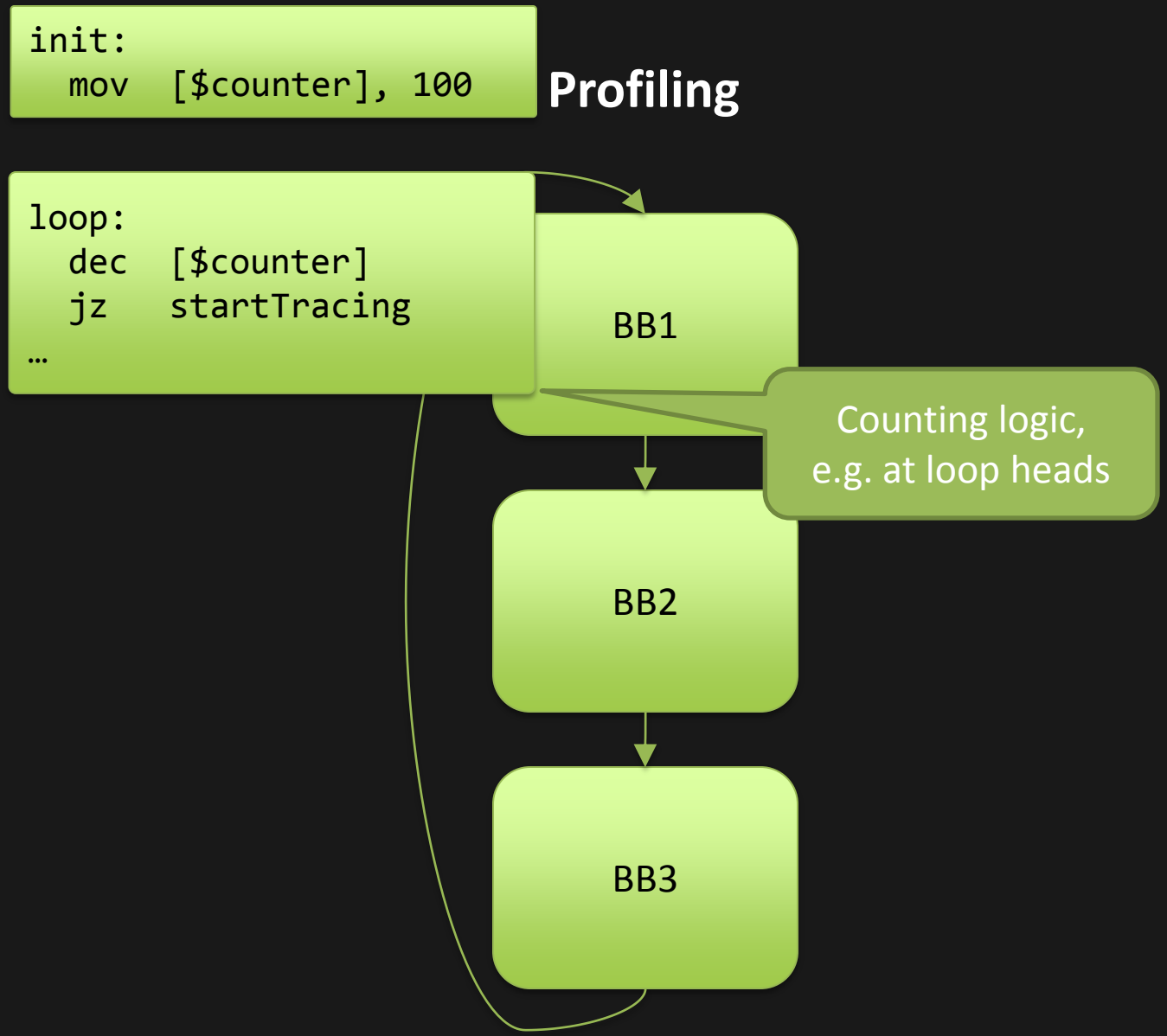
```
int Find(Container c,
         int element)
{
  for (int i = 0;
       i < c.Data.Count;
       i++)
  {
    if (c.Data.GetElement(i) ==
        element)
      return i;
  }
  return -1;
}
```

- Running on my laptop (Core 2 Duo, 2.5 Ghz)
- On 32-bit CLR: 1.20s With Spur: 0.20s => factor of 6

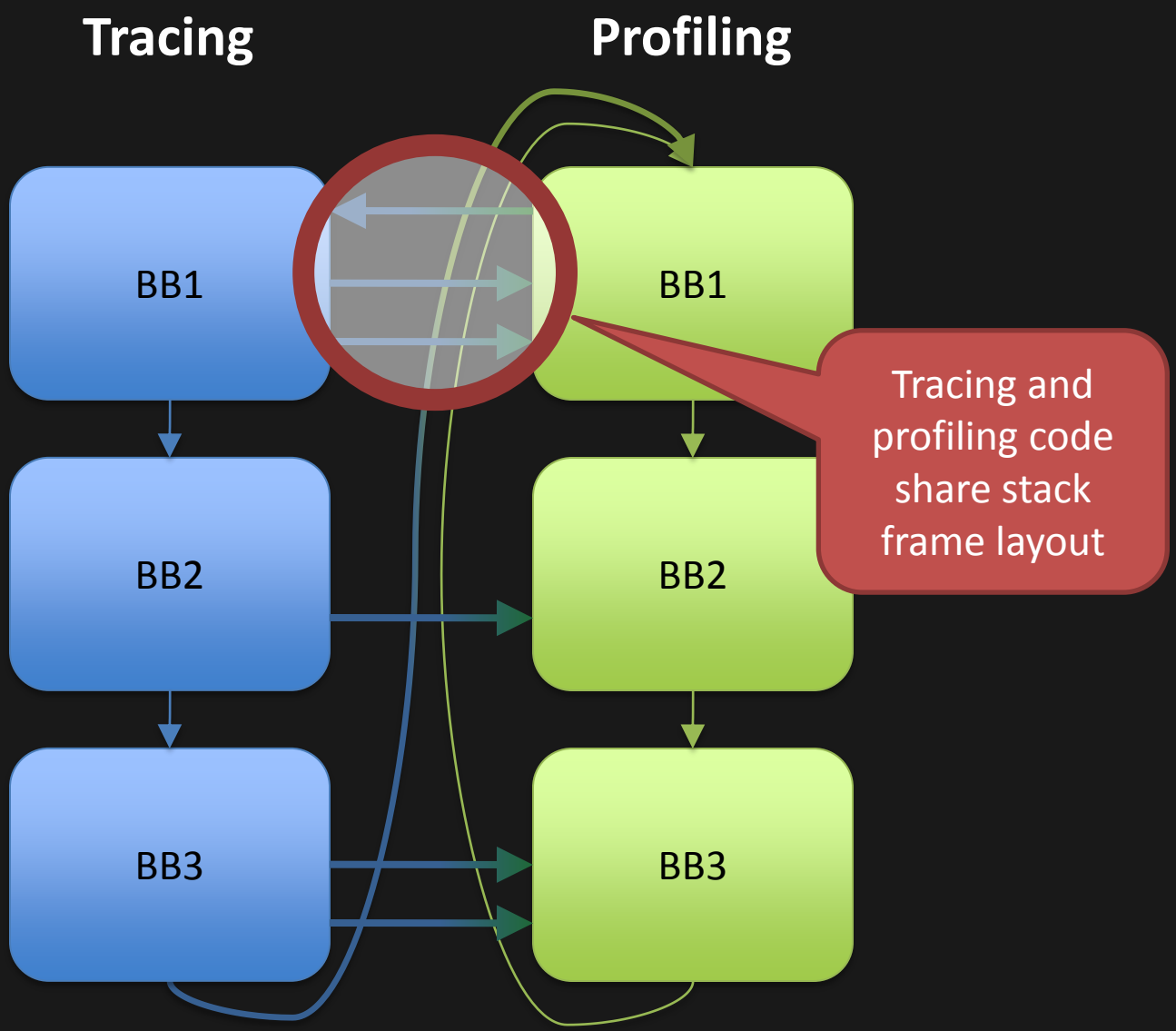
SPUR

UNDER THE HOOD

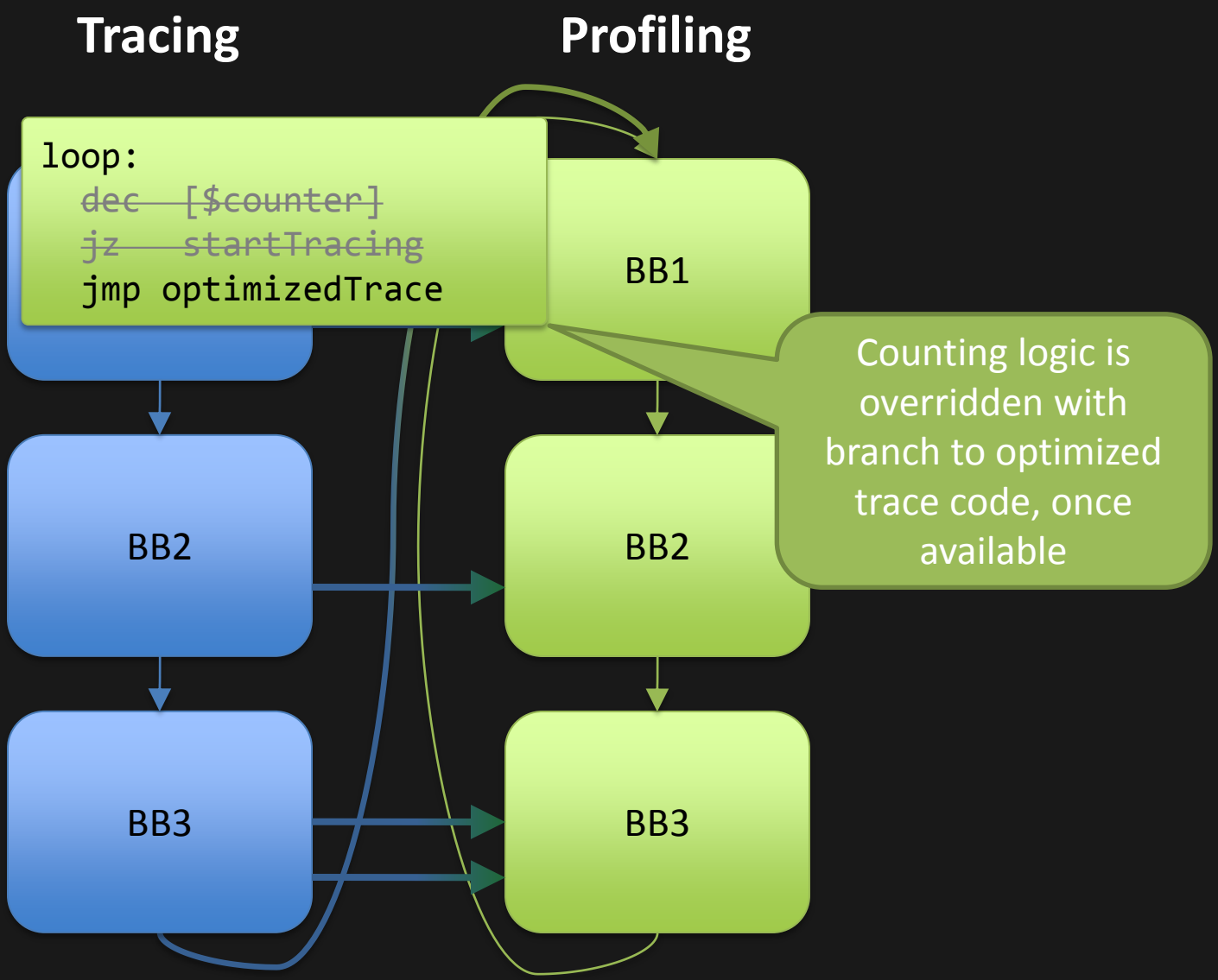
Code Versions in Spur



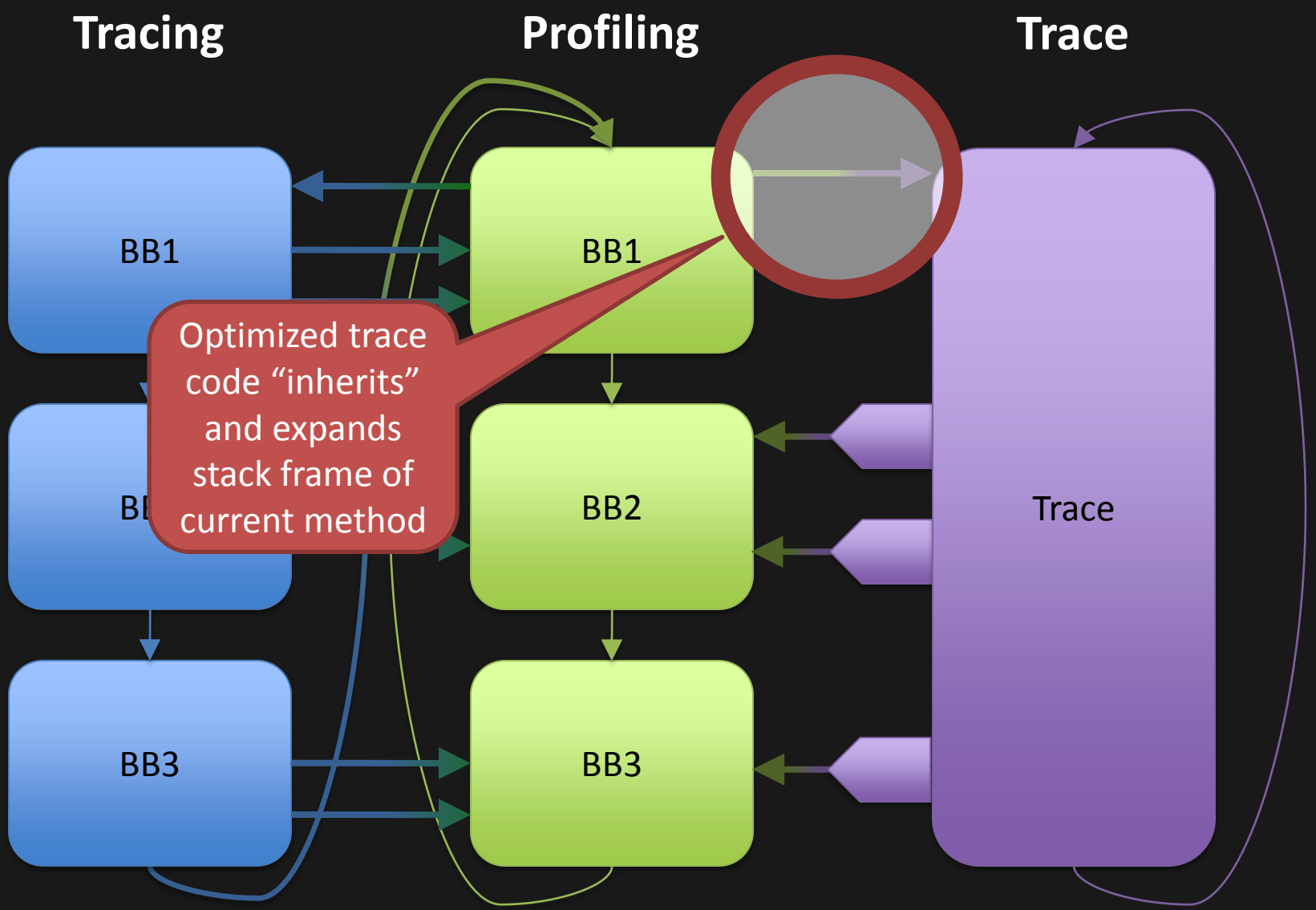
Code Versions in Spur



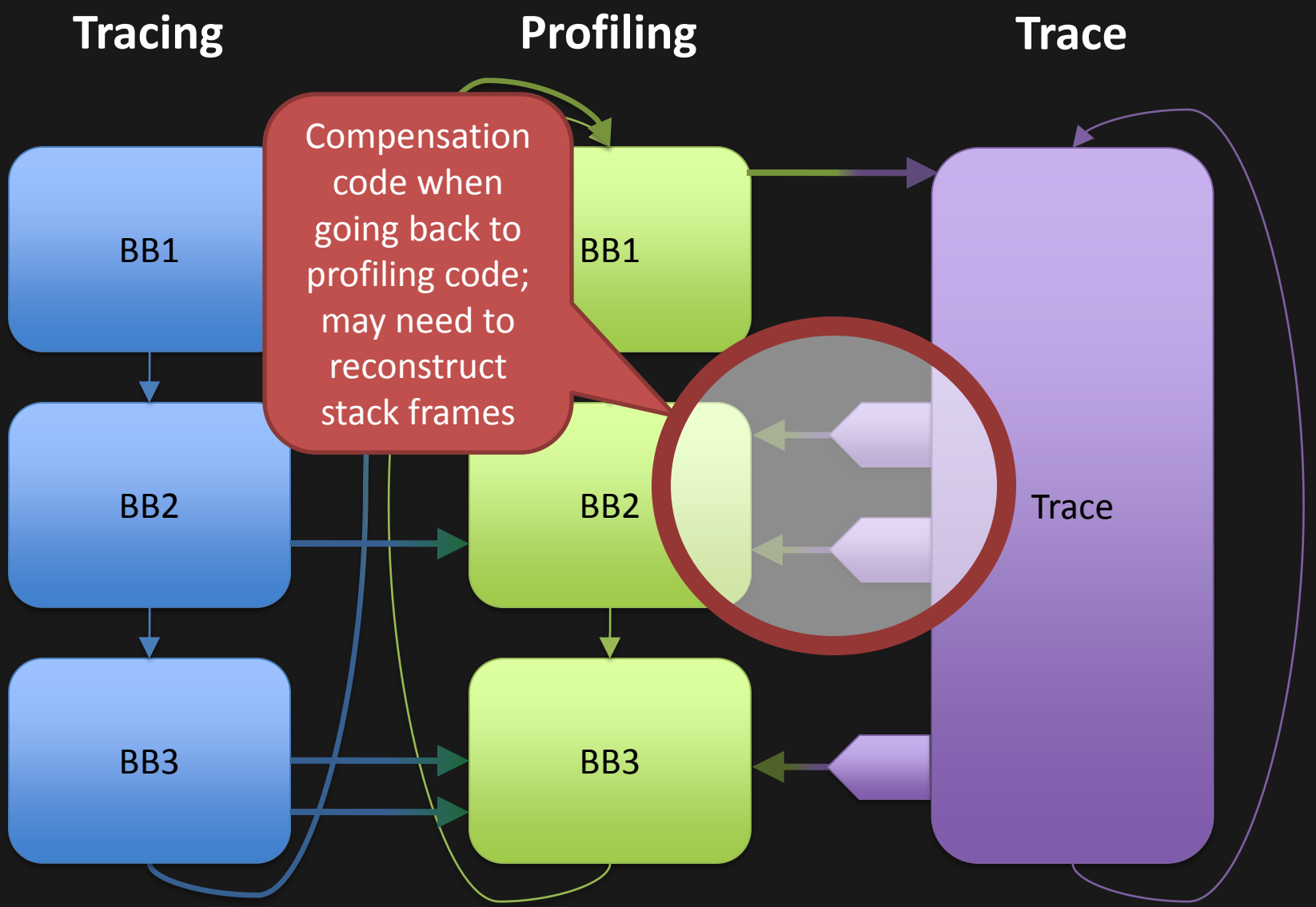
Code Versions in Spur



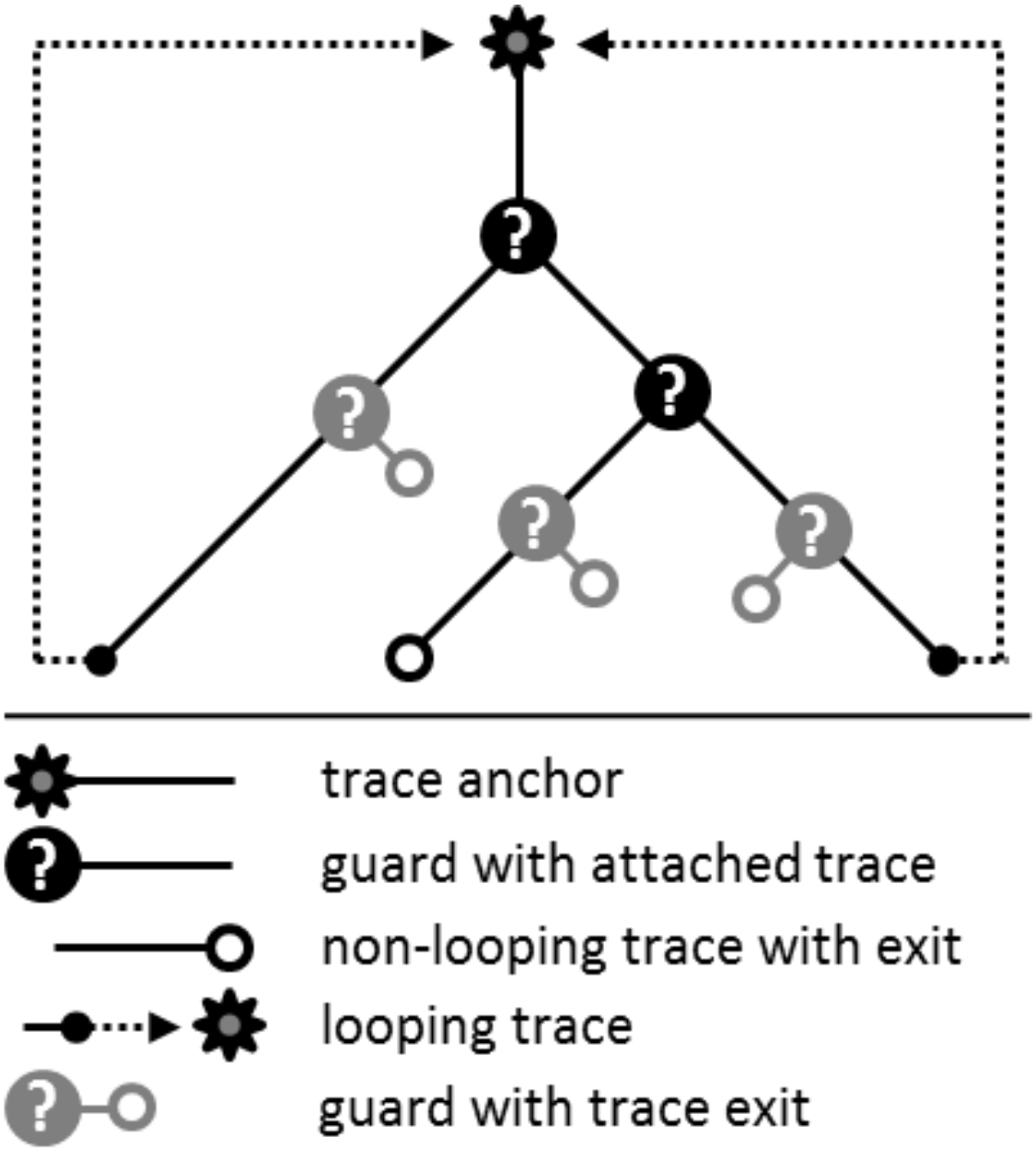
Code Versions in Spur



Code Versions in Spur



Architecture: Trace Trees



SPUR

THE FUTURE

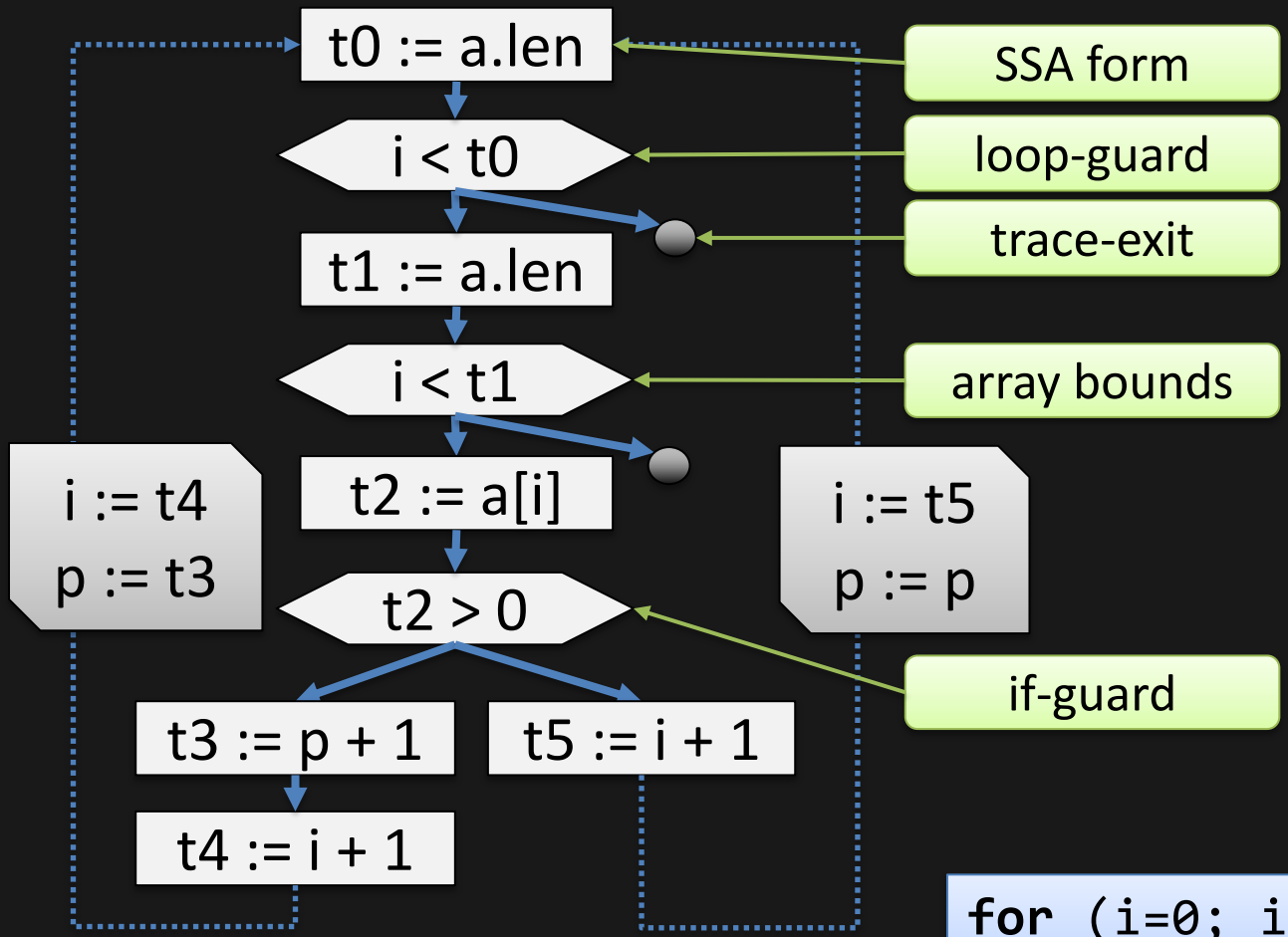
More Research, Potential Applications

- Advanced trace optimization:
  Using automated theorem proving (SMT solvers)
- Automated parallelization and vectorization:
  Leveraging runtime information, overcoming JavaScipt's single-threadedness
- Tracing of C# + JavaScript at the same time:
  Blurring the boundaries of
  Browser Framework and Browser Apps

More Research, Potential Applications

- **Advanced trace optimization:**
  Using automated theorem proving (SMT solvers)
- **Automated parallelization and vectorization:**
  Leveraging runtime information, overcoming JavaScipt's single-threadedness
- **Tracing of C# + JavaScript at the same time:**
  Blurring the boundaries of
  Browser Framework and Browser Apps

Trace Tree Example



Consider this (hot) loop

```
for (i=0; i < a.len; i++)
  if (a[i] > 0) p++;
```

Optimizing Trace Compiler

Standard and not-so-standard optimizations:

- Inlining (for free via tracing)
- Loop unfolding
- Dead Code Eliminations
- Invariant Code Motion
- Constant Folding
- Expression simplification
- Common Subexpression Elimination
- Alias Analysis
- Redundant Guard Elimination
- Redundant Load/Store Elimination
- Speculative Guard Strengthening

Based on pattern matching;

More patterns → more optimizations

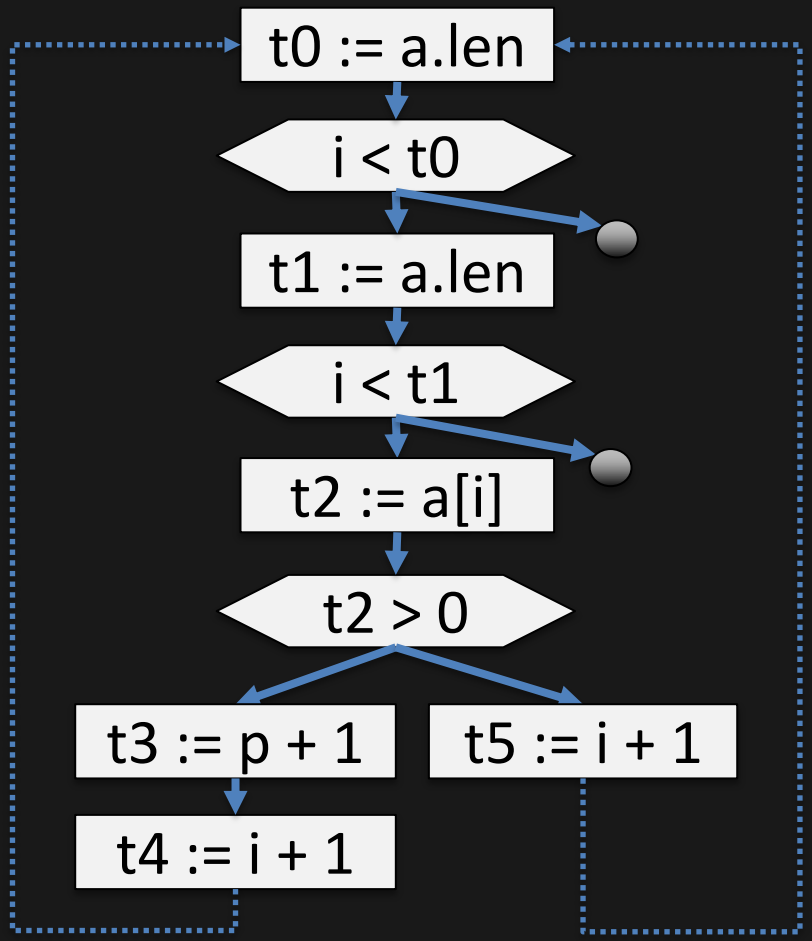
More patterns → more bugs…

Trace Tree Optimizations via SMT Solver

- Correctness burden: compiler -> SMT solver
- Profiling/Tracing: what's worth optimizing
- Enables deep semantic transformations

- Forward Guard Elimination
- Redundant Store Elimination
- Common Subexpression Elimination
  - modulo theories and asserted guards, including alias-analysis and redundant-load elimination.
- Speculative Guard Strengthening

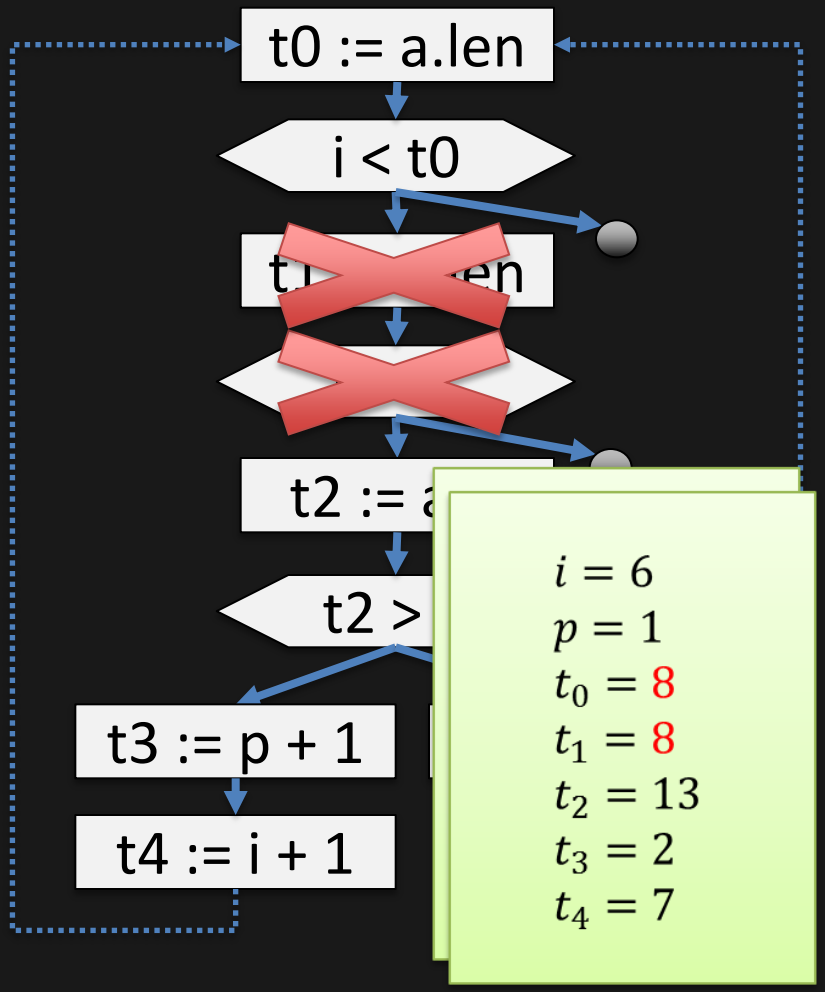
Trace Tree Example



Consider this (hot) loop

```
for (i=0; i < a.len; i++)
  if (a[i] > 0) p++;
```

Trace Tree Optimization Example



$$t_0 = len_0[a]$$

$$bv_<(i, t_0)$$

$$t_1 = len_0[a]$$

$$\neg bv_<(i, t_1)$$

$$t_2 = a[i]$$

$bv_>(t_2, 0)$

$t_3 = bv_+(p, 1)$

$t_4 = bv_+(i, 1)$

$\neg(t_0 = t_4)$

$\neg bv_>(t_2, 0)$

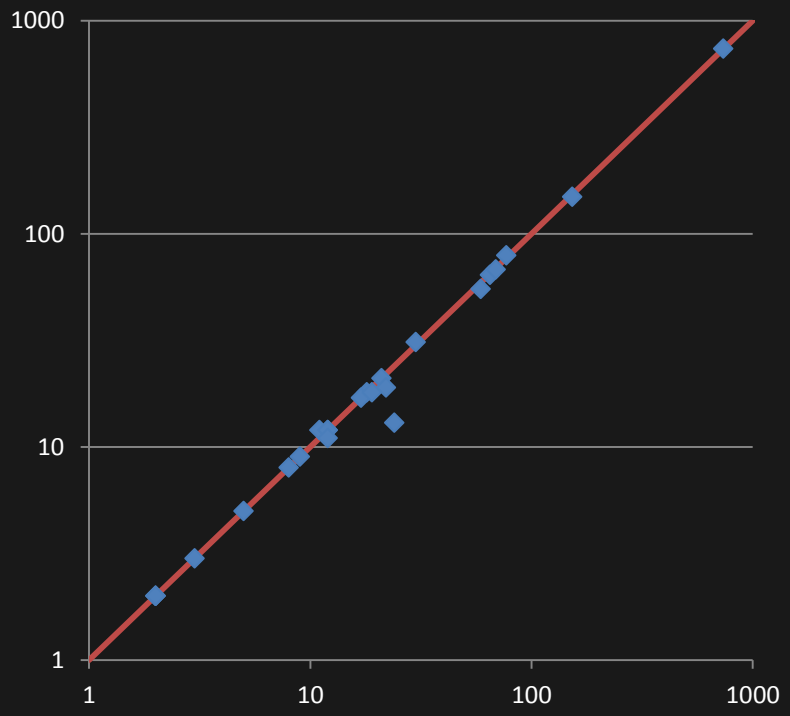
$t_3 = bv_+(p, 1)$

C# benchmarks

program	change in running time	additional instructions removed
Alloc2	-0.8%	3.95%
Cmp	-0.8%	0.00%
Grep	-4.1%	0.36%
Linpack	1.0%	6.43%
Pi	0.0%	7.22%
Sat_solve	-2.2%	2.14%
SciMark	0.6%	7.01%
Sieve	-14.1%	6.46%
Sort	0.5%	0.45%
Wc	-2.0%	0.51%

JavaScript SunSpider benchmarks

program	improvement in running time	additional instructions removed
3d-morph	9%	2.2%
access-binary-trees	3%	1.8%
access-fannkuch	0%	0.6%
access-nsieve	0%	0.5%
bitops-3bit-bits-in-byte	0%	0.0%
bitops-bits-in-byte	0%	0.0%
bitops-bitwise-and	-46%	2.2%
bitops-nsieve-bits	0%	0.5%
controlflow-recursive	0%	0.1%
crypto-sha1	0%	0.3%
math-cordic	0%	3.9%
math-partial-sums	0%	0.8%
math-spectral-norm	0%	0.3%
regexp-dna	0%	0.7%
string-base64	-8%	4.2%
string-fasta	3%	0.8%
string-tagcloud	-3%	1.0%
string-validate-input	-2%	0.3%
3d-cube	-14%	1.0%
access-nbody	0	1.5%
crypto-aes	-5%	1.6%
crypto-md5	0	0.1%
date-format-tofte	-7%	0.4%
date-format-xparb	-1%	1.2%



Automatic Parallelization: Two challenges

We have to know:

1) Is it worthwhile to parallelize?

   Needed: Actual workload, i.e. quantitative data

2) Is it safe to parallelize?

   Needed: Dependency analysis, i.e. read/write., write/write conflicts

Loop Parallelization: Quantified analysis

After having compiled the “optimized” code, analyze if trace parallelization pays off, i.e. if

$$(I*W_e)/P + I*W_i < I*W$$

where

- P parallel threads available
- I number of iterations remaining
- W work per iteration on that trace
- $W \approx (W_e + W_i)$

Parallelization: Splitting

Split loop into 2 phases

Traditional approach: Inspector + Executor

- Inspector checks for dependencies / creates schedule
- Executor performs computations, and mutates memory

Spur also has 2 Phases, but

- Inspector also checks paths conditions of known traces
- If no known trace applies to an iteration, finish parallelization right there, and later record new trace

Parallelization: Example

From Olden benchmarks: Em3d

```
public void computeNewValue() {
    for(int i = 0; i < this.fromCount; i++)
        this.value -= this.coeffs[i] * this.fromNodes[i].value;
}

public void compute() {
    for(int i = this.eNodes.Count - 1; i >= 0; --i)
        this.eNodes[i].computeNewValue();

    for(int i = this.hNodes.Count - 1; i >= 0; --i)
        this.hNodes[i].computeNewValue();
}
```

Preliminary experiments indicate:

- Splitting into Inspector + Executor: 10-20% overhead
- Near linear speed-up by parallelization

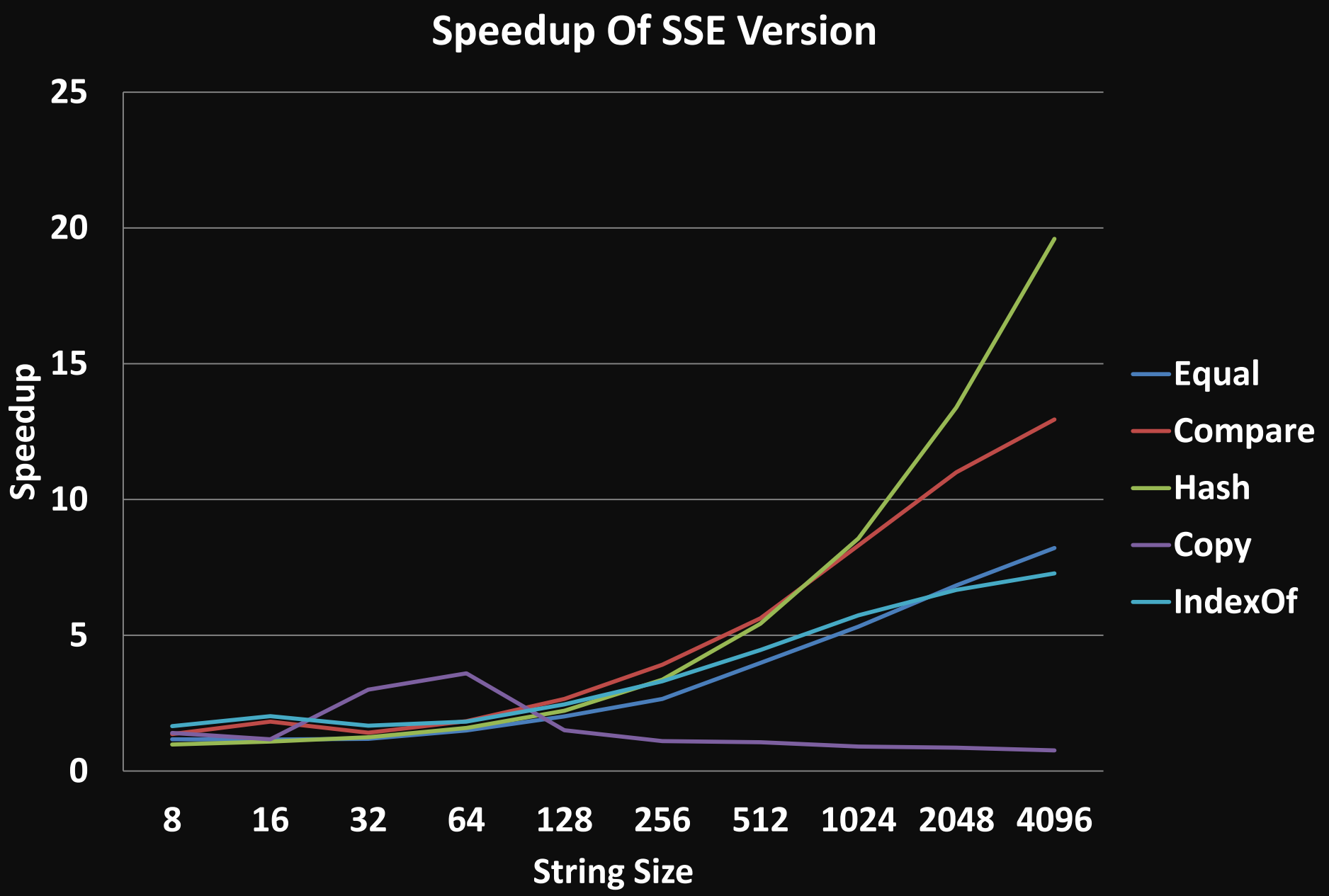
Vectorization

Vectorization is difficult in general JIT setting, as data must be compact and aligned

But applies well to certain library functions:

- System.String
  - Equals
  - GetHashCode
  - CompareTo
  - Copy
  - IndexOf/IndexOfAny

Speed up of System.String functions



Conclusion and Future Work

- One runtime + JIT for all languages on .NET!
- Yields excellent runtime performance
- Research platform

- We are implementing and evaluating
  - Automatic parallelization and vectorization
  - Leveraging SMT solver
  - Blurring boundaries: browser framework vs. apps

http://research.microsoft.com/Spur