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
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"
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
Java Script source compiled to IL, references JavaScript Runtime written in C#

Compiled Script and JavaScript Runtime are JITted to x86 code
Spur traces and optimizes the combination of

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

```javascript
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
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;
}
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)

- SPUR JIT without tracing
- SPUR with CLR JIT (no tracing)
- SPUR with tracing
- TraceMonkey (TM): ships in FireFox browser
- V8: From Google, ships in Chrome browser

-execution time in milliseconds. Smaller is better.
Numbers (SunSpider JavaScript benchmarks)

- Execution time in milliseconds. Smaller is better.

Smiley: SPUR is fastest

- V8
- TM
- Spur with tracing
- SPUR CLR
- SPUR w/o tracing

3d-cube, 3d-morph, 3d-raytrace, acc-bintree, acc-fannkuch, acc-nbody, acc-nsieve, bitops-3bit, bitops-bits, bitops-and, bitops-or, bitops-xor, control-rec, crypto-aes, crypto-md5, date-time, date-xparb, math-cordic, math-partial, math-spectr
Numbers (SunSpider benchmarks cont.)

- Execution time in milliseconds.
  - Smaller is better.

- Graph showing execution times for various benchmarks.
  - Benchmark categories: `regexp-dna`, `string-b64`, `string-fasta`, `string-tagcl`, `string-unpac`, `string-valid`.
  - Tools compared: V8, TM, Spur with tracing, SPUR CLR, SPUR w/o tracing.

- **Execution time in milliseconds.** Smaller is better.
Basic idea: Trace Compilation

```csharp
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#?
```csharp
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));
}
```

---

**SSA form**

**At IL level**
C# equivalent code:

```csharp
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
```

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

- SSA form
- At IL level
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));
}

// 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
C# equivalent code:

```csharp
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

Inlining
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));
}

// 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
    end
}

Invariant Code Motion

SSA form

At IL level
Walkthrough

**C# equivalent code:**

```csharp
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
```
**C# equivalent code:**

```csharp
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)
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;
}
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#: CLR vs. Spur

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

Profiling

init:
  mov  [$counter], 100

loop:
  dec  [$counter]
  jz   startTracing
  ...

Counting logic, e.g. at loop heads
Profiling and Tracing code versions in Spur share stack frame layout.
Code Versions in Spur

Tracing

loop:
  dec[$counter]
  jz startTracing
  jmp optimizedTrace

Profiling

BB1

Counting logic is overridden with branch to optimized trace code, once available
Optimized trace code “inherits” and expands stack frame of current method
Compensation code when going back to profiling code; may need to reconstruct stack frames.
trace anchor

guard with attached trace

non-looping trace with exit

looping trace

guard with trace exit
More Research, Potential Applications

- Advanced trace optimization: Using automated theorem proving (SMT solvers)
- Automated parallelization and vectorization: Leveraging runtime information, overcoming JavaScript’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 JavaScript’s single-threadedness
- Tracing of C# + JavaScript at the same time: Blurring the boundaries of Browser Framework and Browser Apps
for (i=0; i < a.len; i++)
if (a[i] > 0) p++;

\( i < t0 \)
\( i < t1 \)
\( i := t4 \)
\( p := t3 \)
\( t2 := a[i] \)
\( t2 > 0 \)
\( t3 := p + 1 \)
\( t5 := i + 1 \)
\( t4 := i + 1 \)

SSA form
loop-guard
trace-exit
array bounds
if-guard
Consider this (hot) loop
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
for (i=0; i < a.len; i++)
if (a[i] > 0) p++;

Consider this (hot) loop

Trace Tree Example
Trace Tree Optimization Example

\[ t_0 := \text{len}_0[a] \]
\[ b \nu_<(i, t_0) \]
\[ t_1 := \text{len}_0[a] \]
\[ \neg b \nu_<(i, t_1) \]
\[ t_2 := a[i] \]
\[ b \nu_>(t_2, 0) \]
\[ \neg b \nu_>(t_2, 0) \]
\[ t_3 = b \nu_+(p, 1) \]
\[ t_4 = b \nu_+(i, 1) \]
\[ \neg (t_0 = t_4) \]
## C# benchmarks

<table>
<thead>
<tr>
<th>program</th>
<th>change in running time</th>
<th>additional instructions removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc2</td>
<td>-0.8%</td>
<td>3.95%</td>
</tr>
<tr>
<td>Cmp</td>
<td>-0.8%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Grep</td>
<td>-4.1%</td>
<td>0.36%</td>
</tr>
<tr>
<td>Linpack</td>
<td>1.0%</td>
<td>6.43%</td>
</tr>
<tr>
<td>Pi</td>
<td>0.0%</td>
<td>7.22%</td>
</tr>
<tr>
<td>Sat_solve</td>
<td>-2.2%</td>
<td>2.14%</td>
</tr>
<tr>
<td>SciMark</td>
<td>0.6%</td>
<td>7.01%</td>
</tr>
<tr>
<td>Sieve</td>
<td>-14.1%</td>
<td>6.46%</td>
</tr>
<tr>
<td>Sort</td>
<td>0.5%</td>
<td>0.45%</td>
</tr>
<tr>
<td>Wc</td>
<td>-2.0%</td>
<td>0.51%</td>
</tr>
</tbody>
</table>
### JavaScript SunSpider benchmarks

<table>
<thead>
<tr>
<th>program</th>
<th>improvement in running time</th>
<th>additional instructions removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d-morph</td>
<td>9%</td>
<td>2.2%</td>
</tr>
<tr>
<td>access-binary-trees</td>
<td>3%</td>
<td>1.8%</td>
</tr>
<tr>
<td>access-fannkuch</td>
<td>0%</td>
<td>0.6%</td>
</tr>
<tr>
<td>access-nsieve</td>
<td>0%</td>
<td>0.5%</td>
</tr>
<tr>
<td>bitops-3bit-bits-in-byte</td>
<td>0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td>0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>bitops-bitwise-and</td>
<td>-46%</td>
<td>2.2%</td>
</tr>
<tr>
<td>bitops-nsieve-bits</td>
<td>0%</td>
<td>0.5%</td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td>0%</td>
<td>0.1%</td>
</tr>
<tr>
<td>crypto-sha1</td>
<td>0%</td>
<td>0.3%</td>
</tr>
<tr>
<td>math-cordic</td>
<td>0%</td>
<td>3.9%</td>
</tr>
<tr>
<td>math-partial-sums</td>
<td>0%</td>
<td>0.8%</td>
</tr>
<tr>
<td>math-spectral-norm</td>
<td>0%</td>
<td>0.3%</td>
</tr>
<tr>
<td>regexp-dna</td>
<td>0%</td>
<td>0.7%</td>
</tr>
<tr>
<td>string-base64</td>
<td>-8%</td>
<td>4.2%</td>
</tr>
<tr>
<td>string-fasta</td>
<td>3%</td>
<td>0.8%</td>
</tr>
<tr>
<td>string-tagcloud</td>
<td>-3%</td>
<td>1.0%</td>
</tr>
<tr>
<td>string-validate-input</td>
<td>-2%</td>
<td>0.3%</td>
</tr>
<tr>
<td>3d-cube</td>
<td>-14%</td>
<td>1.0%</td>
</tr>
<tr>
<td>access-nbody</td>
<td>0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>crypto-aes</td>
<td>-5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>crypto-md5</td>
<td>0%</td>
<td>0.1%</td>
</tr>
<tr>
<td>date-format-tofte</td>
<td>-7%</td>
<td>0.4%</td>
</tr>
<tr>
<td>date-format-xparb</td>
<td>-1%</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
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
After having compiled the “optimized” code, analyze if trace parallelization pays off, i.e. if

\[ \frac{(I \cdot W_e)}{P} + I \cdot W_i < I \cdot W \]

where

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

```java
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`
- `System.String.CompareTo`
- `System.String.Copy`
- `System.String.IndexOf/System.String.IndexOfAny`
Speed up of System.String functions

Speedup Of SSE Version

String Size

Equal
Compare
Hash
Copy
IndexOf
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