Comparing Verification Condition Generation with Symbolic Execution

Malte Schwerhoff, ETH Zürich
Joint work with Yannis Kassios, Peter Müller

20th December 2012, Dagstuhl
Outline

1. Background
2. Verifiers
3. Benchmark
Background
Modular Verification

- Chalice, a research language developed at Microsoft and ETH
  - Classes and objects
  - Fork-join concurrency with static deadlock detection
  - Asynchronous communication via message passing channels

- User provided specifications
  - Framing via *Implicit Dynamic Frames* (close to Separation Logic)
  - Pre- and postconditions, loop invariants
  - Pure functions for information hiding and specifications
  - Abstract predicates

- Shared data due to *fractional permissions*, monitors and locks
  - `rd(o.f) && acc(o.g, 100%)`
    is read access to `o.f` and write access to `o.g`
Verification condition generation vs. symbolic execution

→ Query prover once per method with all available information

→ Query prover often with limited information
Verifiers
VCG-based Chalice verifier: Chalice

- Encodes a Chalice program in the *intermediate verification language* Boogie

- Heaps and permissions are encoded in Boogie as updateable maps
  - state: \((H, \Pi)\)

- Boogie verifier computes weakest preconditions and uses a prover (currently Z3) to discharge proof obligations
  - correctness criterion: PRE \(\Rightarrow\) wp(BODY, POST)
class Cell {
    var x: int

    function get(): int
        requires rd(x)
        { x }

    method set(y: int)
        requires acc(x)
        ensures acc(x) && get() == y
        { x := y }
}
VCG-Chalice: Example verification

class Cell {
  var x: int

  method set(y: int)
  
  requires acc(x)

  ensures acc(x) && get() == y

  { x := y }

  method get(): int
  
  requires rd(x)

  { x }
}

method client(c: Cell, y: int)
  
  requires acc(c.x)

  ensures acc(c.x) && c.get() == y + 1
  
  { 
    // Π(c,x)=100 ⇒ (Π(c,x)=100 ∧ ∀ u • (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦u],Π,c)=y
    // ⇒ (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1][(c,x)↦u],Π,c)=y+1)))

    // Π(c,x)=100 ∧ (∀ u • (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦u],Π,c)=y)
    // ⇒ (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1][(c,x)↦u],Π,c)=y+1))

    call c.set(y)

    // Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1],Π,c)=y+1

    c.x := c.x + 1

    // Π(c,x)=100 ∧ Cell.get(H[Π,c]=y+1
  }

passed to the prover:
PRE ⇒ WP(BODY, POST)

WP(BODY, POST)

POST
SE-based Chalice verifier: Syxc

- Symbolic execution with a *symbolic state* \((\gamma, h, \pi)\) comprising:
  - *Store* \(\gamma\) mapping local variables to symbolic values (*terms*)
  - *Symbolic heap* \(h\) consisting of *heap chunks*
    \[
    r.f \mapsto v \# p
    \]
    “field \(r.f\) has the value \(v\) and we have \(p\) permissions to the field”
  - *Path condition* \(\pi\) with assumptions such as \(v > 0, r \neq \text{null}\)

- Verifier manages \(h\), discharges permission (spatial) assertions
- Prover manages \(\pi\), discharges boolean assertions

- Idea: Divide the work, gain performance
class Cell {
    var x: int

    method set(y: int)
        requires acc(x)
        ensures acc(x) && get() == y
        { x := y }
    
    function get(): int
        requires rd(x)
        { x }
}

method client(c: Cell, y: int)
    requires acc(c.x)
    ensures acc(c.x) && c.get() == y + 1
    {
        // γ: {c ↦ tc, y ↦ ty}
        // h: {tc.x ↦ tx # 100}
        // π: {} 
        call c.set(y)
        
        // h: {tc.x ↦ tx' # 100}
        // π: {Cell.get(tc, tx') == ty ∧ Cell.get(tc, tx) == tx'}
        c.x := c.x + 1
        
        // h: {tc.x ↦ tx' + 1 # 100}
        // π: {Cell.get(tc, tx') == ty ∧ Cell.get(tc, tx) == tx'} ∧ Cell.get(tc, tx' + 1) == tx + 1
    }

Verifier finds write access to `c.x` in `h`

Verifier finds write access to `c.x` in `h`

Verifier finds write access to `c.x` in `h`

Prover proves `π ⇒ c.get() == y + 1`
Syxc: State separation consequences

- Separation between heap and path conditions has consequences

```java
method client(x: Cell, y: Cell)

exec

requires rd(x.f) && rd(y.f)
{
    if (x == y) {
        assert x.f == y.f
    }
}
```
Syxc: State separation consequences

- Separation between heap and path conditions has consequences

```java
method client(x: Cell, y: Cell)
  requires rd(x.f) && rd(y.f)
{
  if (x == y) {
    assert x.f == y.f
  }
}
```
Syxc: State separation consequences

- Separation between heap and path conditions has consequences

```java
method client(x: Cell, y: Cell)
    requires rd(x.f) && rd(y.f)
{
    if (x == y) {
        assert x.f == y.f
        // Would fail naively because
        // tv == tw is unknown to the prover
    }
}
```

- Heap compression: Modify $h$ according to $\pi$
Syxc: State separation consequences

- Separation between heap and path conditions has consequences

```java
method client(x: Cell, y: Cell)
    requires rd(x.f) && rd(y.f)
{
    if (x == y) {
        assert x.f == y.f
            // Holds
    }
}
```

- **Heap compression**: Modify $h$ according to $\pi$
- $O(n^2)$ calls to the prover
- Newly added equalities require fix-point iteration: $O(n^3)$
Syxc: Branching

- Symbolic execution branches on conditionals → possible explosion of the number of paths

```java
method client(x: Cell, b: bool)
  requires b ==> acc(x.f) // Impure conditional
{ ... }
```

- Two execution paths with different \( h \) and \( \pi \)
  1. \( h \) contains chunk \( tx.f \mapsto tv \# 100 \), \( \pi \) contains \( b \)
  2. \( h \) does not contain such a chunk, \( \pi \) contains \( \neg b \)
Syxc: Branching

Detected while benchmarking, Syxc was been 4x slower

The precondition is pure, i.e., it does not affect the heap

- Encode it as a pure implication \( tb \Rightarrow ti > 0 \)
- Add it to \( \pi \), continue without branching (but the prover still needs to branch)
- Decreases verification time significantly
- Our assumption: Additional knowledge allows the prover to optimise

```java
method client(b: bool, i: int)
    requires b ==> i > 0 // Pure conditional
{ ... }
```
Syxc: Other separation consequences

```java
method client(x: Cell, y: Cell)
    requires acc(x.f, 60%) && acc(y.f, 60%)
{ assert x != y }
```

- Information in $h$ influence $\pi$

- Heap chunks suppress uncertainty
  - Empty $\pi$, prover reports ‘unknown’ for both ‘$b$’ and ‘$\neg b$’
  - Empty collection (e.g., $h$), verifier reports ‘false’ for both ‘contains $b$/\neg b$’
  - Sound for permissions because they cannot be negated
  - Unsound when modelling other properties via heap chunks
Benchmark
Setup

- Benchmarked
  - 22 succeeding and 7 failing tests from the Chalice test suite
  - 66 seeded failing test cases

- Comparable setup: same parser, same Z3 version & settings

- Average statistics over ten runs per tool and test case
Metrics

- **Execution time** (seconds)

- **Quantifier instantiations** in the prover
  - The less QIs a method provokes, the more predictable/robust it is
  - Extreme example (the *matching loop*):
    
    $\forall x : \text{int} :: f(x) == f(g(x))$

- **Conflicts** generated by the prover
  - Roughly: the more conflicts a prover causes, the more proof space it explores
## Results

- Average performance of Syxc relative to Chalice

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>QI</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hold (22)</td>
<td>32% (8% – 47%)</td>
<td>5% (0.2% – 19%)</td>
<td>31% (4% – 192%)</td>
</tr>
<tr>
<td>Fail (73)</td>
<td>31% (3% – 43%)</td>
<td>9% (0.2% – 85%)</td>
<td>39% (0% – 334%)</td>
</tr>
</tbody>
</table>
Results

- Average performance of Syxc relative to Chalice

<table>
<thead>
<tr>
<th></th>
<th>RT</th>
<th>QI</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hold (22)</td>
<td>32% (8% – 47%)</td>
<td>5% (0.2% – 19%)</td>
<td>31% (4% – 192%)</td>
</tr>
<tr>
<td>Fail (73)</td>
<td>31% (3% – 43%)</td>
<td>9% (0.2% – 85%)</td>
<td>39% (0% – 334%)</td>
</tr>
<tr>
<td>Hold &gt; 100/75/25%</td>
<td>0/0/17</td>
<td>0/0/0</td>
<td>1/1/9</td>
</tr>
<tr>
<td>Fail &gt; 100/75/25%</td>
<td>0/0/60</td>
<td>0/1/5</td>
<td>8/11/24</td>
</tr>
</tbody>
</table>
Conclusions

- Test suite is still rather small, results have to be taken with caution!

- Results indicate that in the case of VCG-Chalice vs. Syxc
  - SE is faster (runtime)
  - SE is more predictive (QIs)
  - SE tends to be more focused (conflicts)

- The number of QIs is significantly smaller due to state separation into heap and path conditions
  - Less information for the prover, less non-goal-directed work
  - Separation of labour between verifier and prover seems beneficial

- However, limited information might not always be beneficial, as hinted at by the branching problem
Current and upcoming work

- Syxc
  - Immutable (frozen) data structures
  - Class and trait inheritance
  - Debugger

- Syxc retires, reuse experience and target Scala
  - Immutability
  - Inheritance
  - Actor concurrency
  - ...

- Tool chain: Round tripping between SE Verifier, AI Inferer, Slicer
The End
Backup Slides
Chalice: a Cell example and access permissions

```java
class Cell {
    var x: int

    function get(): int
        requires rd(x)
    { x }

    method set(y: int)
        requires acc(x)
        ensures acc(x) && get() == y
    { x := y }
}
```

- General access permissions: `acc(o.x, p)`, where 0<p<=100
  - p<100 grants read access
  - p==100 grants write access
  - sum of all permissions to o.x at any given time: 100
  - rd(x) grants infinitesimal permission \( \varepsilon \)
class Cell {
    var x: int

    predicate P { acc(x) }

    function get(): int requires rd(P) { unfolding rd(P) in x }

    method set(y: int) requires P ensures P && get() == y {
        unfold P
        x := y
        fold P
    }

    method clone() returns (c: Cell) requires rd(P) ensures rd(P) && c.P && c.get() == get() {
        c := new Cell
        fold c.P
        call c.set(get())
    }
}

class Client {
    method fails(c: Cell) requires c.P {
        fork tk1 := c.set(1)
        fork tk2 := c.set(2) // ERR
    }

    method succeeds(c: Cell) requires c.P {
        fork tk1 := c.clone()
        fork tk2 := c.clone()
    }
}
Tools

- Latest Chalice version uses a new permission model not yet supported by Syxc, hence we had to use a slightly outdated Chalice version
- Syxc uses Chalice (as a library) to parse input into an AST
- Recent Boogie version; limited to one error per Chalice method
- Z3 3.1, smtlib2 frontend via std-io, interaction is logged in a file
- Syxc uses nearly the same Z3 settings as Chalice does, except
  - Syxc requires Z3 to respond to every command, not only to `check-sat`
  - Syxc uses global declarations, not scope-local ones
- Other differences:
  - Syxc encodes snapshots, references and lists as Z3 integers → might increase the number of quantifier instantiations
  - Syxc uses Boogie’s sequence axiomatisation, but they range over integers only, whereas Boogie’s are polymorphic → might increase the workload for Z3
A closer look at the 9 cases where Syxc causes more conflicts

- 1 holds, 8 fail
- 6 out of these 9 are variations (seeded) of only 2 original programs
- The 9 cases perform relatively poor in general
  - 7 Runtimes above average
  - 6 Quantifier instantiations above average
## Results

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
<th>Meth.</th>
<th>Syxc</th>
<th>Chalice</th>
<th>Syxc rel. to Chalice</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVLTree.iterative</td>
<td>212</td>
<td>3</td>
<td>1.96</td>
<td>15.1</td>
<td>12.98 0.18 11.93</td>
</tr>
<tr>
<td>AVLTree.nokeys</td>
<td>572</td>
<td>19</td>
<td>13.25 2320.2 1895.9</td>
<td>171.45 2424408 19455</td>
<td>7.73 0.1 9.75</td>
</tr>
<tr>
<td>Cell</td>
<td>131</td>
<td>11</td>
<td>1.62 235 41.2</td>
<td>5.62 9736 536</td>
<td>28.83 2.41 7.69</td>
</tr>
<tr>
<td>CopyLessMessagePassing</td>
<td>54</td>
<td>3</td>
<td>1.38 217 33</td>
<td>3.71 4779 210</td>
<td>37.2 4.54 15.71</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a</td>
<td>62</td>
<td>3</td>
<td>1.55 773 70</td>
<td>3.89 4014 181</td>
<td>39.85 19.26 38.67</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a2</td>
<td>66</td>
<td>3</td>
<td>1.55 852 74</td>
<td>4.19 10492 323</td>
<td>36.99 8.12 22.91</td>
</tr>
<tr>
<td>Dining-philosophers</td>
<td>74</td>
<td>3</td>
<td>1.55 1765.6 128</td>
<td>4.49 17144 416</td>
<td>34.52 10.3 30.77</td>
</tr>
<tr>
<td>Iterator</td>
<td>123</td>
<td>6</td>
<td>2.59 398.2 606.9</td>
<td>5.47 7577 316</td>
<td>47.35 5.26 192.06</td>
</tr>
<tr>
<td>Iterator2</td>
<td>111</td>
<td>6</td>
<td>1.56 188 37</td>
<td>5.17 12867 304</td>
<td>30.17 1.46 12.17</td>
</tr>
<tr>
<td>LoopLockChange</td>
<td>69</td>
<td>5</td>
<td>1.17 204 38</td>
<td>3.7 2632 207</td>
<td>31.62 7.75 18.36</td>
</tr>
<tr>
<td>OwickiGries</td>
<td>31</td>
<td>2</td>
<td>1.16 124 41</td>
<td>3.34 5727 206</td>
<td>34.73 2.17 19.9</td>
</tr>
<tr>
<td>PetersonsAlgorithm</td>
<td>70</td>
<td>3</td>
<td>1.88 1247 269.2</td>
<td>11.05 166647 1504</td>
<td>17.01 0.75 17.9</td>
</tr>
<tr>
<td>ProdConsChannel</td>
<td>78</td>
<td>6</td>
<td>1.37 279 117.4</td>
<td>4.26 4648 317</td>
<td>32.16 6 37.03</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>171</td>
<td>11</td>
<td>1.72 515.8 138.4</td>
<td>7.71 17293 584</td>
<td>22.31 2.98 23.7</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>28</td>
<td>1</td>
<td>1 36 25</td>
<td>2.65 1101 106</td>
<td>37.74 3.27 23.58</td>
</tr>
<tr>
<td>RockBand</td>
<td>109</td>
<td>13</td>
<td>1.03 22 14</td>
<td>2.63 258 28</td>
<td>39.16 8.53 50</td>
</tr>
<tr>
<td>Sieve</td>
<td>56</td>
<td>4</td>
<td>1.52 830 177</td>
<td>6.72 21938 689</td>
<td>22.62 3.78 25.69</td>
</tr>
<tr>
<td>Swap</td>
<td>19</td>
<td>2</td>
<td>0.99 74 21</td>
<td>2.58 877 60</td>
<td>38.37 8.44 35</td>
</tr>
<tr>
<td>Prog1</td>
<td>33</td>
<td>4</td>
<td>0.97 25 4</td>
<td>2.4 196 12</td>
<td>40.42 12.76 33.33</td>
</tr>
<tr>
<td>Prog2</td>
<td>52</td>
<td>7</td>
<td>1.24 79 14</td>
<td>4.31 9138 341</td>
<td>28.77 0.86 4.11</td>
</tr>
<tr>
<td>Prog3</td>
<td>163</td>
<td>16</td>
<td>1.35 314 92.1</td>
<td>3.78 7596 249</td>
<td>35.71 4.13 36.99</td>
</tr>
<tr>
<td>Prog4</td>
<td>19</td>
<td>1</td>
<td>0.91 10 3</td>
<td>2.35 261 20</td>
<td>38.72 3.83 15</td>
</tr>
</tbody>
</table>

- Longest method (iterative); longest example (nokeys)
- Best runtime (13%) and quantifier instantiation (0.1%)
## Results

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
<th>Meth.</th>
<th>Syxc Time (s)</th>
<th>QI</th>
<th>Conflicts</th>
<th>Chalice Time (s)</th>
<th>QI</th>
<th>Conflicts</th>
<th>Syxc rel. to Chalice Time (%)</th>
<th>QI (%)</th>
<th>Conflicts (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVLTree.iterative</td>
<td>212</td>
<td>3</td>
<td>1.96</td>
<td>263</td>
<td>153.6</td>
<td>15.1</td>
<td>146362</td>
<td>1287</td>
<td>12.98</td>
<td>0.18</td>
<td>11.93</td>
</tr>
<tr>
<td>AVLTree.nokeys</td>
<td>572</td>
<td>19</td>
<td>13.25</td>
<td>2320.2</td>
<td>1895.9</td>
<td>171.45</td>
<td>2424408</td>
<td>19455</td>
<td>7.73</td>
<td>0.1</td>
<td>9.75</td>
</tr>
<tr>
<td>Cell</td>
<td>131</td>
<td>11</td>
<td>1.62</td>
<td>235</td>
<td>41.2</td>
<td>5.62</td>
<td>9736</td>
<td>536</td>
<td>28.83</td>
<td>2.41</td>
<td>7.69</td>
</tr>
<tr>
<td>CopyLessMessagePassing</td>
<td>54</td>
<td>3</td>
<td>1.38</td>
<td>217</td>
<td>33</td>
<td>3.71</td>
<td>4779</td>
<td>210</td>
<td>37.2</td>
<td>4.54</td>
<td>15.71</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a</td>
<td>62</td>
<td>3</td>
<td>1.55</td>
<td>773</td>
<td>70</td>
<td>3.89</td>
<td>4014</td>
<td>181</td>
<td>39.85</td>
<td>19.26</td>
<td>38.67</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a2</td>
<td>66</td>
<td>3</td>
<td>1.55</td>
<td>852</td>
<td>74</td>
<td>4.19</td>
<td>10492</td>
<td>323</td>
<td>36.99</td>
<td>8.12</td>
<td>22.91</td>
</tr>
<tr>
<td>Dining-philosophers</td>
<td>74</td>
<td>3</td>
<td>1.55</td>
<td>1765.6</td>
<td>128</td>
<td>4.49</td>
<td>17144</td>
<td>416</td>
<td>34.52</td>
<td>10.3</td>
<td>30.77</td>
</tr>
<tr>
<td>Iterator</td>
<td>123</td>
<td>6</td>
<td>2.59</td>
<td>398.2</td>
<td>606.9</td>
<td>5.47</td>
<td>7577</td>
<td>316</td>
<td>47.35</td>
<td>5.26</td>
<td>192.06</td>
</tr>
<tr>
<td>Iterator2</td>
<td>111</td>
<td>6</td>
<td>1.56</td>
<td>188</td>
<td>37</td>
<td>5.17</td>
<td>12867</td>
<td>304</td>
<td>30.17</td>
<td>1.46</td>
<td>12.17</td>
</tr>
<tr>
<td>LoopLockChange</td>
<td>69</td>
<td>5</td>
<td>1.17</td>
<td>204</td>
<td>38</td>
<td>3.7</td>
<td>2632</td>
<td>207</td>
<td>31.62</td>
<td>7.75</td>
<td>18.36</td>
</tr>
<tr>
<td>OwickiGries</td>
<td>31</td>
<td>2</td>
<td>1.16</td>
<td>124</td>
<td>41</td>
<td>3.34</td>
<td>5727</td>
<td>206</td>
<td>34.73</td>
<td>2.17</td>
<td>19.9</td>
</tr>
<tr>
<td>PetersonsAlgorithm</td>
<td>70</td>
<td>3</td>
<td>1.88</td>
<td>1247</td>
<td>269.2</td>
<td>11.05</td>
<td>166647</td>
<td>1504</td>
<td>17.01</td>
<td>0.75</td>
<td>17.9</td>
</tr>
<tr>
<td>ProdConsChannel</td>
<td>78</td>
<td>6</td>
<td>1.37</td>
<td>279</td>
<td>117.4</td>
<td>4.26</td>
<td>4648</td>
<td>317</td>
<td>32.16</td>
<td>6</td>
<td>37.03</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>171</td>
<td>11</td>
<td>1.72</td>
<td>515.8</td>
<td>138.4</td>
<td>7.71</td>
<td>17293</td>
<td>584</td>
<td>22.31</td>
<td>2.98</td>
<td>23.7</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>25</td>
<td>2.65</td>
<td>1101</td>
<td>106</td>
<td>37.74</td>
<td>3.27</td>
<td>23.58</td>
</tr>
<tr>
<td>RockBand</td>
<td>109</td>
<td>13</td>
<td>1.03</td>
<td>22</td>
<td>14</td>
<td>2.63</td>
<td>258</td>
<td>28</td>
<td>39.16</td>
<td>8.53</td>
<td>50</td>
</tr>
<tr>
<td>Sieve</td>
<td>56</td>
<td>4</td>
<td>1.52</td>
<td>830</td>
<td>177</td>
<td>6.72</td>
<td>21938</td>
<td>689</td>
<td>22.62</td>
<td>3.78</td>
<td>25.69</td>
</tr>
<tr>
<td>Swap</td>
<td>19</td>
<td>2</td>
<td>0.99</td>
<td>74</td>
<td>21</td>
<td>2.58</td>
<td>877</td>
<td>60</td>
<td>38.37</td>
<td>8.44</td>
<td>35</td>
</tr>
<tr>
<td>Prog1</td>
<td>33</td>
<td>4</td>
<td>0.97</td>
<td>25</td>
<td>4</td>
<td>2.4</td>
<td>196</td>
<td>12</td>
<td>40.42</td>
<td>12.76</td>
<td>33.33</td>
</tr>
<tr>
<td>Prog2</td>
<td>52</td>
<td>7</td>
<td>1.24</td>
<td>79</td>
<td>14</td>
<td>4.31</td>
<td>9138</td>
<td>341</td>
<td>28.77</td>
<td>0.86</td>
<td>4.11</td>
</tr>
<tr>
<td>Prog3</td>
<td>163</td>
<td>16</td>
<td>1.35</td>
<td>314</td>
<td>92.1</td>
<td>3.78</td>
<td>7596</td>
<td>249</td>
<td>35.71</td>
<td>4.13</td>
<td>36.99</td>
</tr>
<tr>
<td>Prog4</td>
<td>19</td>
<td>1</td>
<td>0.91</td>
<td>10</td>
<td>3</td>
<td>2.35</td>
<td>261</td>
<td>20</td>
<td>38.72</td>
<td>3.83</td>
<td>15</td>
</tr>
</tbody>
</table>

- Similar behaviour for test case PetersonsAlgorithm: Runtime speedup above average, very low QI ratio
### Results

<table>
<thead>
<tr>
<th>File</th>
<th>Syxc</th>
<th>Chalice</th>
<th>Syxc rel. to Chalice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>LOC</td>
<td>Meth.</td>
<td>Time (s)</td>
</tr>
<tr>
<td>AVLTree.iterative</td>
<td>212</td>
<td>3</td>
<td>1.96</td>
</tr>
<tr>
<td>AVLTree.nokeys</td>
<td>572</td>
<td>19</td>
<td>13.25</td>
</tr>
<tr>
<td>Cell</td>
<td>131</td>
<td>11</td>
<td>1.62</td>
</tr>
<tr>
<td>CopyLessMessagePassing</td>
<td>54</td>
<td>3</td>
<td>1.38</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a</td>
<td>62</td>
<td>3</td>
<td>1.55</td>
</tr>
<tr>
<td>CopyLessMessagePassing-w/a2</td>
<td>66</td>
<td>3</td>
<td>1.55</td>
</tr>
<tr>
<td>Dining philosophers</td>
<td>74</td>
<td>2</td>
<td>1.35</td>
</tr>
<tr>
<td><strong>Iterator</strong></td>
<td>123</td>
<td>6</td>
<td>2.59</td>
</tr>
<tr>
<td>Iterator2</td>
<td>111</td>
<td>6</td>
<td>1.56</td>
</tr>
<tr>
<td>LoopLockChange</td>
<td>69</td>
<td>5</td>
<td>1.17</td>
</tr>
<tr>
<td>OwickiGries</td>
<td>31</td>
<td>2</td>
<td>1.16</td>
</tr>
<tr>
<td>PetersonsAlgorithm</td>
<td>70</td>
<td>3</td>
<td>1.88</td>
</tr>
<tr>
<td>ProdConsChannel</td>
<td>78</td>
<td>6</td>
<td>1.37</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>171</td>
<td>11</td>
<td>1.72</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>28</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RockBand</td>
<td>109</td>
<td>13</td>
<td>1.03</td>
</tr>
<tr>
<td>Sieve</td>
<td>56</td>
<td>4</td>
<td>1.52</td>
</tr>
<tr>
<td>Swap</td>
<td>19</td>
<td>2</td>
<td>0.99</td>
</tr>
<tr>
<td>Prog1</td>
<td>33</td>
<td>4</td>
<td>0.97</td>
</tr>
<tr>
<td>Prog2</td>
<td>52</td>
<td>7</td>
<td>1.24</td>
</tr>
<tr>
<td>Prog3</td>
<td>163</td>
<td>16</td>
<td>1.35</td>
</tr>
<tr>
<td>Prog4</td>
<td>19</td>
<td>1</td>
<td>0.91</td>
</tr>
</tbody>
</table>

- Conflicts are less conclusive:
  - **Iterator:** conflicts above average, QI below average, still twice as fast