# Automatic verification of counter systems via domain-specific multi-result supercompilation

Andrei V. Klimov  Ilya G. Klyuchnikov  Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

2012-07 / Meta 2012

# Outline

1. SC + filtering/selection $\Longrightarrow$ analysis/verification
2. Domain-specific supercompilation (DSSC): what are the benefits?
3. Multi-result supercompilation (MRSC): selecting the best results
4. DSSC + MRSC $\Longrightarrow$ synergistic effect: less CPU and memory resources
5. Conclusions

# Outline

1. SC + filtering/selection $\Longrightarrow$ analysis/verification
2. Domain-specific supercompilation (DSSC): what are the benefits?
3. Multi-result supercompilation (MRSC): selecting the best results
4. DSSC + MRSC $\Longrightarrow$ synergistic effect: less CPU and memory resources
5. Conclusions

# SC + filtering/selection $\Longrightarrow$ analysis/verification

Suppose that

- `sc` is a supercompiler such that `sc p` is semantically equivalent to `p`.
- `good` is a program checker (a human or an algorithm).
  (`good p = true` means that the program `p` is "good".)

Let us construct a "problem solver".

- Problem: let `p` be such that `good p = false`.
- Supercompilation: `sc p = p'`.
- Checking: `good p' = true`. (Thus `p'` is "more understandable" than `p`).
- Automation:
  `let p' = sc p in if good p' then Just p' else Nothing`.

## Conclusion

SC + filtering/selection $\Longrightarrow$ analysis/verification

# MESI protocol: its model in form of a counter system

Initial states:

$$(i, 0, 0, 0)$$

Transitions:

$$\begin{array}{lcl}
(i, e, s, m) \mid i \geq 1 & \longrightarrow & (i-1, 0, s+e+m+1, 0) \\
(i, e, s, m) \mid e \geq 1 & \longrightarrow & (i, e-1, s, m+1) \\
(i, e, s, m) \mid s \geq 1 & \longrightarrow & (i+e+s+m-1, 1, 0, 0) \\
(i, e, s, m) \mid i \geq 1 & \longrightarrow & (i+e+s+m-1, 1, 0, 0)
\end{array}$$

Unsafe states:

$$\begin{array}{l}
(i, e, s, m) \mid m \geq 2 \\
(i, e, s, m) \mid s \geq 1 \wedge m \geq 1
\end{array}$$

# MESI protocol: its model in form of a Refal program (1)

```
*$MST_FROM_ENTRY;
*$STRATEGY Applicative;
*$LENGTH 0;

$ENTRY Go {e.A (e.I) =
  <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}

Loop {
  () (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Result (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>;
  (s.A e.A) (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Loop (e.A)
    <RandomAction s.A
    (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>>;}

Result{
(Invalid e.1)(Modified s.2 e.2)(Shared s.3 e.3)(Exclusive e.4) = False;
(Invalid e.1)(Modified s.21 s.22 e.2)(Shared e.3)(Exclusive e.4) = False;
(Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) = True;}

...
```

# MESI protocol: its model in form of a Refal program (2)

```
...

RandomAction {
* rh Trivial
* rm
A (Invalid s.1 e.1) (Modified e.2) (Shared e.3) (Exclusive e.4) =
    (Invalid e.1) (Modified ) (Shared s.1 e.2 e.3 e.4 ) (Exclusive );
* wh1 Trivial
* wh2
B (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive s.4 e.4) =
    (Invalid e.1)(Modified s.4 e.2)(Shared e.3)(Exclusive e.4);
* wh3
C (Invalid e.1)(Modified e.2)(Shared s.3 e.3)(Exclusive e.4) =
    (Invalid e.4 e.3 e.2 e.1)(Modified )(Shared )(Exclusive s.3);
* wm
D (Invalid s.1 e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    (Invalid e.4 e.3 e.2 e.1)(Modified )(Shared )(Exclusive s.1);
}
```

# MESI protocol: the residual Refal program (1)

```
* InputFormat: <Go e.41 >
$ENTRY Go {
  (e.101 ) = True ;
  A e.41 (s.103 e.101 ) = <F24 (e.41 ) (e.101 ) s.103 > ;
  D e.41 (s.104 e.101 ) = <F35 (e.41 ) (e.101 ) s.104 > ;}

* InputFormat: <F24 (e.109 ) (e.110 ) s.111 e.112 >
F24 {
  () (e.110 ) s.111 e.112 = True ;
  (A e.109 ) (s.114 e.110 ) s.111 e.112 =
    <F24 (e.109 ) (e.110 ) s.114 s.111 e.112 > ;
  (C e.109 ) (e.110 ) s.111 e.112 =
    <F35 (e.109 ) (e.110 ) s.111 e.112 >;
  (D e.109 ) (s.115 e.110 ) s.111 e.112 =
    <F35 (e.109 ) (s.111 e.112 e.110) s.115 > ;}

...
```

# MESI protocol: the residual Refal program (2)

```
...

* InputFormat: <F35 (e.109 ) (e.110 ) s.111 e.112 >
F35 {
  () (e.110 ) s.111 e.112 = True ;
  (A e.109 ) (e.110 ) s.111 s.118 e.112 =
    <F24 (e.109 ) (e.112 e.110 ) s.118 s.111 > ;
  (A e.109 ) (s.119 e.110 ) s.111 = <F24 (e.109 ) (e.110 ) s.119 s.111 >;
  (B ) (e.110 ) s.111 e.112 = True ;
  (B A e.109 ) (e.110 ) s.111 s.125 e.112 =
    <F24 (e.109 ) (e.112 e.110 ) s.125 s.111 > ;
  (B A e.109 ) (s.126 e.110 ) s.111 =
    <F24 (e.109 ) (e.110 ) s.126 s.111> ;
  (B D e.109 ) (e.110 ) s.111 s.127 e.112 =
    <F35 (e.109 ) (s.111 e.112 e.110) s.127 > ;
  (B D e.109 ) (s.128 e.110 ) s.111 =
    <F35 (e.109 ) (s.111 e.110 ) s.128> ;
  (D e.109 ) (e.110 ) s.111 s.120 e.112 =
    <F35 (e.109 ) (s.111 e.112 e.110) s.120 > ;
  (D e.109 ) (s.121 e.110 ) s.111 = <F35 (e.109 ) (s.111 e.110 ) s.121 >;}
```

# MESI protocol: the residual Refal program (3)

## Thesis

The residual program is unable to return `False`.

## Justification

(1) The symbol `False` does not appear in the program.
(2) Refal programs do not produce new symbols dynamically.

## Insufficiency of the above justification

Refal is dynamically typed. Thus `False` can leak in via the input data! This trick is known as "injection" (and is very popular with hackers).

Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*. ACM, New York, NY, USA, 372–382.
http://doi.acm.org/10.1145/1111037.1111070

# MESI protocol: the residual Refal program (4)

## A solution

Residual programs can be submitted to a data flow analysis algorithm.

Neil D. Jones and Nils Andersen. 2007. Flow analysis of lazy higher-order functional programs. Theor. Comput. Sci. 375, 1–3 (April 2007), 120–136.
`http://dx.doi.org/10.1016/j.tcs.2006.12.030`

Is the game worth the candle? Yes, for example, it makes sense under the two following conditions.

1. We have to consider a lot of residual programs (hundreds or thousands). In this case the analysis has to be automated.
2. The algorithm `good` is smart enough to "understand" `sc p`, but is unable to "understand" `p`. Namely, `good (sc p)` is true, but `good p` is false.

# Weaknesses of general-purpose supercompilation

A general-purpose supercompiler cannot be used as a “black-box”.

- The representation of data has to conform to subtle details of the internal machinery of SCP4, rather than comply with the problem domain. (For example, natural numbers are represented by strings of symbols, and their addition by string concatenation.)
- Input programs have to be supplemented with some directions (in form of comments) for SCP4, thereby providing SCP4 with certain information about the problem domain. Thus again the user needs to understand the internals of SCP4.

The problem of correctness.

- To what extent can we trust the results produced by SCP4? The internals of SCP4 are complicated and the source code is big. Thus the problem of formally verifying SCP4 seems to be intractable.

# Outline

# Domain-specific supercompilation

Abstractly speaking, suppose we have:

- A domain-specific language.
- A domain-specific supercompilation algorithm.

Hence, we can throw a nice

**Slogan**

"Domain-specific supercompilation for domain-specific languages!"

Why? What for? What are potential benefits?

# DSSC: advantages

- Input tasks can be written in a domain-specific language. Hence, in a more natural way.
- The machinery of supercompilation can be simplified.
  - The supercompiler is easier to implement.
  - The correctness is easier to prove.
- Exploiting the specifics of the problem domain.
  - Specific data structures.
  - Specific operations.
  - Some mathematical properties of the operations are known in advance.
- Some classes of problems can be shown to be solvable by supercompilation.

Here is an example of a simplified supercompiler that was formally verified.

Dimitur Krustev. A simple supercompiler formally verified in Coq. In *Second International Workshop on Metacomputation in Russia*, 2010.

# DSSC: counter systems (Klimov)

The supercompilation algorithm can be be simplified in the following ways.

- Configurations have the form $(a_1, \ldots, a_n)$, where $a_i$ is either a natural number $N$ or the symbol $\omega$ (a wildcard, representing an arbitrary natural number).
- Driving deals only with tests of the form either $e = N$ or $e \geq N$, where $e$ is an arithmetic expression and $N$ a natural number.
- Arithmetic expressions can only contain the operators $+$, $-$, natural numbers and the symbol $\omega$.

Thus

- There are no nested function calls.
- Generalization of configurations is performed by replacing some numbers $N$ with $\omega$.

Andrei Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Ershov Informatics Conference, volume 7162 of LNCS, pages 193–209, 2011.

# MESI protocol: the DSL program

```
object MESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0)
  val rules: List[TransitionRule] = List(
    {case List(i, e, s, m) if i>=1 => List(i-1, 0, s+e+m+1, 0)},
    {case List(i, e, s, m) if e>=1 => List(i, e-1, s, m+1)},
    {case List(i, e, s, m) if s>=1 => List(i+e+s+m-1, 1, 0, 0)},
    {case List(i, e, s, m) if i>=1 => List(i+e+s+m-1, 1, 0, 0)})
  def unsafe(c: Conf) = c match {
    case List(i, e, s, m) if m>=2 => true
    case List(i, e, s, m) if s>=1 && m>=1 => true
    case _ => false
  }
}
```

Notes.

- The DSL program is non-deterministic, and is rather close to the informal specification of the protocol model.
- The DSL is implemented atop of the language Scala by means of “embedding”.
- The repetition of `case List List(i, e, s, m)` could have been eliminated, in order to make the DSL more “human-friendly”.

# DSL for counter systems: implementation

```
package object counters {
  type Conf = List[Expr]
  type TransitionRule = PartialFunction[Conf, Conf]
  ...
}

sealed trait Expr { ... }

trait Protocol {
  val start: Conf
  val rules: List[TransitionRule]
  def unsafe(c: Conf): Boolean
}
```

Notes.

- A DSL program is a mixture of first-order values (numbers, lists) and higher-order values (functions). This trick is known as “shallow embedding”.
- A transition rule is a partial function, since a rule can be inapplicable to a configuration.

# Domain-specific residualization

Specifics of ~~optimizing~~ analyzing supercompilation

- Input DSL programs are supposed to be analyzed, rather than executed.
- The results produced by supercompilation are meant for subsequent analysis, rather than for execution.

Thus, suppose `sc` is an ~~optimizing~~ analyzing supercompiler, and `p` an input program.

- There is no good reason for `p` and `sc p` to be written in the same programming language.
- There is no good reason for `sc p` to be written in a programming language!
- It seems to be a good idea to have `sc` produce scripts for another formal verification system or a proof assistant (such as Isabelle or Coq).

The supercompiler for counter systems does produce scripts for the proof-assistant Isabelle!

# MESI protocol: the script for Isabelle (1)

Andrei V. Klimov, Ilya G. Klyuchnikov, Sergei A. Romanenko Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit Preprint No. 24. Keldysh Institute of Applied Mathematics, Moscow, 2012.
`http://library.keldysh.ru/preprint.asp?lg=e&id=2012-24`

A script is a collection of inductive predicate definitions + a few lemmas/theorems (with proofs). It can be executed by Isabelle automatically, without human assistance.

```
theory mesi
imports Main
begin

inductive unsafe :: "(nat * nat * nat * nat) => bool" where
  "unsafe (i, e, s, Suc (Suc m))" |
  "unsafe (i, e, Suc s, Suc m)"

...
```

# MESI protocol: the script for Isabelle (2)

The non-trivial (and non-standard) part of the script is the definition of the relation `mesi'` that is weaker than `mesi`. In contrast to `mesi`, the definition of `mesi'` is not recursive! Without this hint Isabelle would be unable to prove the safeness of all reachable states.

```
...

inductive mesi :: "(nat * nat * nat * nat) => bool" where
  "mesi (i, 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i, 0, Suc (s + e + m), 0)" |
  "mesi (i, Suc e, s, m) ==> mesi (i, e, s, Suc m)" |
  "mesi (i, e, Suc s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)"

inductive mesi' :: "(nat * nat * nat * nat) => bool" where
  "mesi'(_, Suc 0, 0, 0)" |
  "mesi'(_, 0, 0, Suc 0)" |
  "mesi'(_, 0, _, 0)"

...
```

# MESI protocol: the script for Isabelle (3)

And now here are the proofs.

```
...

lemma inclusion: "mesi c ==> mesi' c"
  apply(erule mesi.induct)
  apply(erule mesi'.cases | simp add: mesi'.intros)+
done

lemma safety: "mesi' c ==> ~unsafe c"
  apply(erule mesi'.cases)
  apply(erule unsafe.cases | auto)+
done

theorem valid: "mesi c ==> ~unsafe c"
  apply(insert inclusion safety, simp)
done

end
```

# Outline

# Variations of supercompilation

Variations of supercompilation.

- The classic/deterministic/single-result (functions):
  `sc p = r`.
- Non-deterministic (relations):
  `p ndsc r`.
- Multi-result (multi-valued functions):
  `mrsc p = [`$r_1$`,...,`$r_k$`]`.

# A variety of candidates $\Longrightarrow$ the search for best solutions

The search for a single solution.

- ```
  solve p =
      let p' = sc p in
      if good p' then Just p' else Nothing
  ```

The search for all solutions.

- `solve p = filter good (mrsc p)`

The selection of best solutions.

- `solve p = filter best (filter good (mrsc p))`

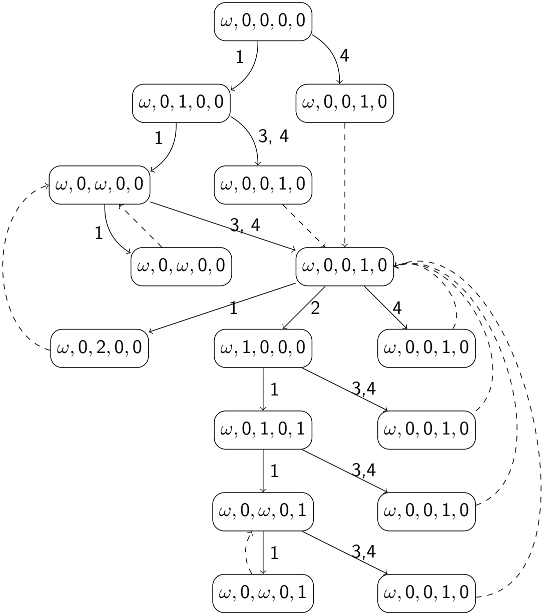
This is a simplification. Usually, `best` is a binary relation, rather than a predicate.
  ```

# MOESI protocol: the DSL program

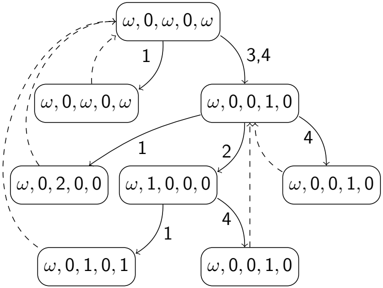
```
case object MOESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0, 0)
  val rules: List[TransitionRule] =
    List({ // rm
      case List(i, m, s, e, o) if i>=1 =>
        List(i-1, 0, s+e+1, 0, o+m)
    }, { // wh2
      case List(i, m, s, e, o) if e>=1 =>
        List(i, m+1, s, e-1, o)
    }, { // wh3
      case List(i, m, s, e, o) if s+o>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    }, { // wm
      case List(i, m, s, e, o) if i>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    })

  def unsafe(c: Conf) = c match {
    case List(i, m, s, e, o) if m>=1 && e+s+o>=1 => true
    case List(i, m, s, e, o) if m>=2 => true
    case List(i, m, s, e, o) if e>=2 => true
    case _ => false
  }
}
```

# MOESI protocol: the graph (SC)



# MOESI protocol: the minimal graph (MRSC)



# MOESI protocol: graph size reduction. What is the trick?

### "A sudden flash of inspiration"

The initial configuration $(\omega, 0, 0, 0, 0)$ can be immediately generalized to $(\omega, 0, \omega, 0, \omega)$!

Such "crazy" generalizations are not performed by a single-result optimizing supercompiler. Why?

- Generalization leads to loss of information. Hence, it should be avoided by all means.
- Generalization is only performed by necessity, when the whistle blows.
- When optimizing a loop, it makes sense to partially unroll the loop, even if, finally, the supercompiler has to fall into the general case.
- Postponing a generalization is likely to improve the execution speed. Code bloat is considered to be "a lesser evil". (Modulo memory caches. . . )

However, this is not of importance in the case of analyzing supercompilation.

# Graph sizes for a number of protocols

| | SC | MRSC |
|---|---|---|
| Synapse | 11 | 6 |
| MSI | 8 | 6 |
| MOSI | 26 | 14 |
| MESI | 14 | 9 |
| MOESI | 20 | 9 |
| Illinois | 15 | 13 |
| Berkley | 17 | 6 |
| Firefly | 12 | 10 |
| Futurebus | 45 | 24 |
| Xerox | 22 | 13 |
| Java | 35 | 25 |
| ReaderWriter | 48 | 9 |
| DataRace | 9 | 5 |

# Outline

# DSSC + MRSC: synergy $\Longrightarrow$ resource savings

The search for best solutions.

- `solve p = filter best (filter good (mrsc p))`

A three-pass algorithm! What about "fusing" them? How? Early filtering!

- Let us filter graphs, rather than residual programs.
- Let us filter (even) incomplete graphs.

This can be done by taking into account the specifics of the problem domain.

- The predicate `unsafe` is monotonic with respect to generalization of configurations.
- The (multi-result) supercompilation algorithm can only remove a configuration by replacing it with a more general configuration.

**So, "unsafeness" is monotonic with respect to graph building**

If an unsafe configuration appears in a graph $G$, all graphs derived from $G$ are bound to contain unsafe configurations. Therefore, $G$ can be discarded without losing any solution.

# Pruning: a classic idea from artificial intelligence

A yet another property of the (multi-result) supercompilation algorithm.

- All graphs derived from a graph $G$ cannot be lesser in size than $G$.

## So, pruning!

If there has been found a complete graph $G$, consisting of safe configurations, all incomplete graphs exceeding $G$ in size can be discarded without losing the minimal solution. (Just because their descendants would be greater in size than $G$.)

D. Poole and A. K. Mackworth. Artificial Intelligence - Foundations of Computational Agents. Cambridge University Press, 2010.

# Taking into account the properties of generalization

Notation: $c \sqsubseteq c' \iff$ the configuration $c'$ is not less general than the configuration $c$.

## Definition

$c'$ is a one-step generalization of a configuration $c$, if $c'$ can be obtained from $c$ by replacing a numeric component of $c$ with $\omega$.

## The structure of the set of generalizations of a configuration

If $c \sqsubseteq c'$, then there exists a sequence of generalizations $c_1, \ldots, c_k$, such that $c = c_1$, $c' = c_k$ and $c_{i+1}$ is a one-step generalization of $c_i$.

## Example

$(0,0) \sqsubseteq (\omega,0) \sqsubseteq (\omega,\omega)$
$(0,0) \sqsubseteq (0,\omega) \sqsubseteq (\omega,\omega)$

# 5 variations of the supercompiler (MRSC)

- SC1. A graph is examined by the filter only after having been completed. Thus, no use is made of the knowledge about domain-specific properties of generalization or the predicate `unsafe`.
- SC2. SC1 + when rebuilding a configuration, only one-step generalizations are considered (all other generalizations remain reachable by a number of steps).
- SC3. SC2 + the configurations produced by generalization are checked for being safe, and the unsafe ones are immediately discarded.
- SC4. SC3 + the configurations that could be produced by driving a configuration $c$ are checked for being safe. If one or more of the new configurations turn out to be unsafe, driving is not performed for $c$.
- SC5. SC4 + the current graph is discarded if there has been already constructed a complete graph that is smaller in size than the current graph (pruning).

# Resources consumed by 5 supercompilers (1)

| | | SC1 | SC2 | SC3 | SC4 | SC5 |
|---|---|---|---|---|---|---|
| Synapse | completed | 48 | 37 | 3 | 3 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 2 |
| | commands | 321 | 252 | 25 | 25 | 15 |
| MSI | completed | 22 | 18 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 1 |
| | commands | 122 | 102 | 15 | 15 | 12 |
| MOSI | completed | 1233 | 699 | 6 | 6 | 1 |
| | pruned | 0 | 0 | 0 | 0 | 5 |
| | commands | 19925 | 11476 | 109 | 109 | 35 |
| MESI | completed | 1627 | 899 | 6 | 3 | 1 |
| | pruned | 0 | 0 | 27 | 20 | 21 |
| | commands | 16329 | 9265 | 211 | 70 | 56 |
| MOESI | completed | 179380 | 60724 | 81 | 30 | 2 |
| | pruned | 0 | 0 | 0 | 24 | 36 |
| | commands | 2001708 | 711784 | 922 | 384 | 126 |

# Resources consumed by 5 supercompilers (2)

| | | SC1 | SC2 | SC3 | SC4 | SC5 |
|---|---|---|---|---|---|---|
| Illinois | completed | 2346 | 1237 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 21 | 17 | 18 |
| | commands | 48364 | 26636 | 224 | 74 | 61 |
| Berkley | completed | 3405 | 1463 | 30 | 30 | 2 |
| | pruned | 0 | 0 | 0 | 0 | 14 |
| | commands | 26618 | 12023 | 282 | 282 | 56 |
| Firefly | completed | 2503 | 1450 | 2 | 2 | 1 |
| | pruned | 0 | 0 | 2 | 2 | 3 |
| | commands | 39924 | 24572 | 47 | 25 | 21 |
| Futurebus | completed | - | - | - | - | 4 |
| | pruned | - | - | - | - | 148328 |
| | commands | - | - | - | - | 516457 |
| Xerox | completed | 317569 | 111122 | 29 | 29 | 2 |
| | pruned | 0 | 0 | 0 | 0 | 1 |
| | commands | 5718691 | 2031754 | 482 | 482 | 72 |

# Resources consumed by 5 supercompilers (3)

| | | SC1 | SC2 | SC3 | SC4 | SC5 |
|---|---|---|---|---|---|---|
| Java | completed | - | - | - | - | 10 |
| | pruned | - | - | - | - | 329886 |
| | commands | - | - | - | - | 1043563 |
| ReaderWriter | completed | 892371 | 402136 | 898 | 898 | 6 |
| | pruned | 0 | 0 | 19033 | 19033 | 1170 |
| | commands | 24963661 | 11872211 | 123371 | 45411 | 3213 |
| DataRace | completed | 51 | 39 | 8 | 8 | 3 |
| | pruned | 0 | 0 | 0 | 0 | 4 |
| | commands | 360 | 279 | 57 | 57 | 31 |

# Outline

1. SC + filtering/selection $\Longrightarrow$ analysis/verification
2. Domain-specific supercompilation (DSSC): what are the benefits?
3. Multi-result supercompilation (MRSC): selecting the best results
4. DSSC + MRSC $\Longrightarrow$ synergistic effect: less CPU and memory resources
5. Conclusions

# Conclusions

- The benefits of domain-specific supercompilation.
  - Problems/tasks can be formulated in a natural way (DSL).
  - Some knowledge about the problem domain can be built into the supercompiler.
  - The machinery of supercompilation can be simplified (by removing redundant "gears").
  - The correctness of simplified supercompilation is easier to ensure.
- The benefits of multi-result supercompilation.
  - More modular structure of the supercompiler (decoupling the whistle and the generalization algorithm).
  - The search and selection of best solutions.
- Domain-specific SC + multi-result SC $\Longrightarrow$ a synergistic effect.
  - Search space reduction (by several orders of magnitude).