Towards Persistent and Parallel Asynchronous Adaptive Specialisation for Generic Data-Parallel Array Processing in SAC

Clemens Grelck, Heinz Wiesinger

Universiteit van Amsterdam

25th Symposium on
Implementation and Application of Functional Languages
Nijmegen, Netherlands
August 30, 2013

What is Generic Array Processing ?

- Programming with multidimensional arrays
- Abstract from structural properties of arrays
- Treat arrays as abstract values
- Storage, layout, operations, ... all implicit

Single Assignment C (SAC)

Language:

- Purely functional programming language
- Generic data-parallel array processing
- Syntax imitates ISO C
  - Assignment sequences
  - Branches
  - Loops
- Semantics as expected by naive (imperative) programmer

Single Assignment C (SAC)

Language:

- Purely functional programming language
- Generic data-parallel array processing
- Syntax imitates ISO C
  - Assignment sequences
  - Branches
  - Loops
- Semantics as expected by naive (imperative) programmer

Compiler:

- Highly optimising compiler
- Performance competitive with Fortran or C
- Automatic parallelisation for
  - Symmetric multicore multiprocessors
  - Graphics accelerators
  - Heterogeneous systems

Multidimensional Array Calculus

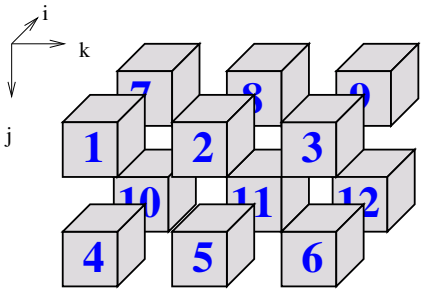
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

dim:	2
shape:	[3,3]
data:	[1,2,3,4,5,6,7,8,9]

Multidimensional Array Calculus

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

dim: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]

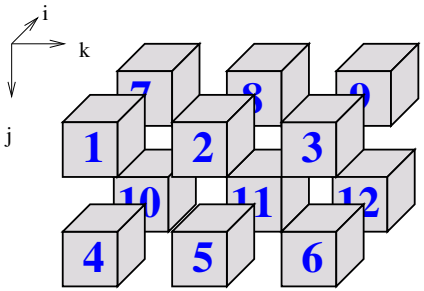


dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

Multidimensional Array Calculus

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

dim: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]



dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

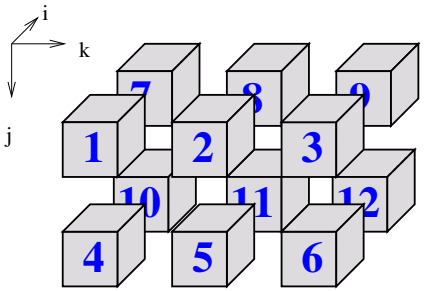
[1, 2, 3, 4, 5, 6]

dim: 1
shape: [6]
data: [1,2,3,4,5,6]

Multidimensional Array Calculus

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

dim: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]



dim: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

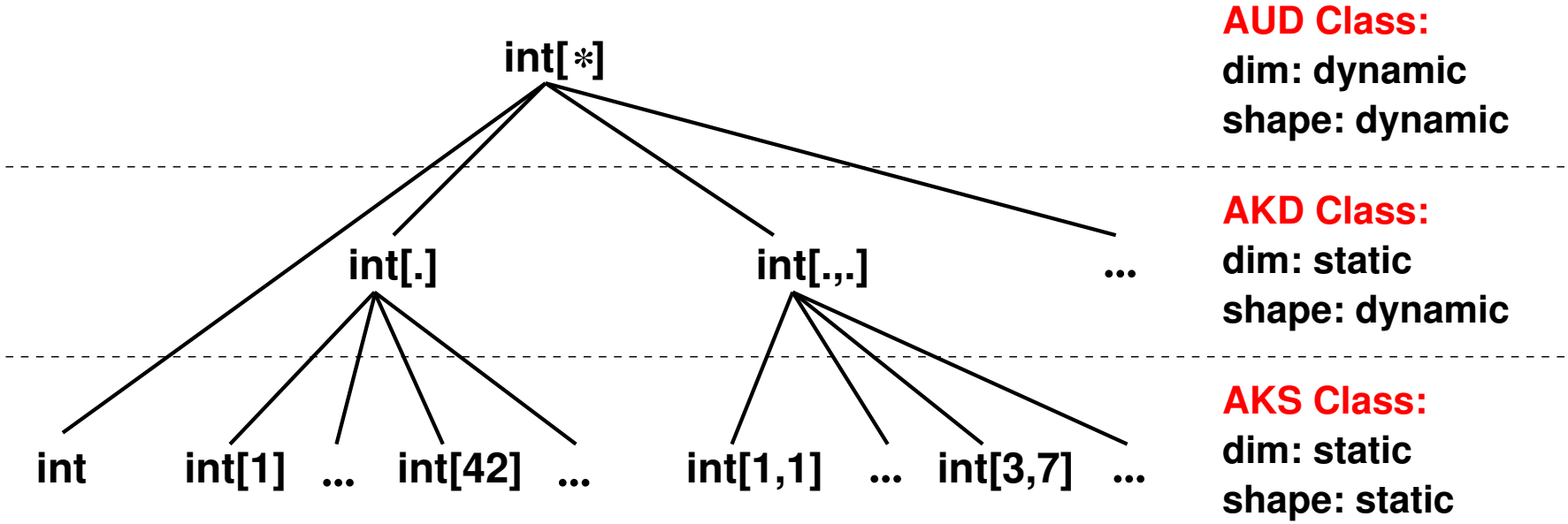
[1, 2, 3, 4, 5, 6]

dim: 1
shape: [6]
data: [1,2,3,4,5,6]

42

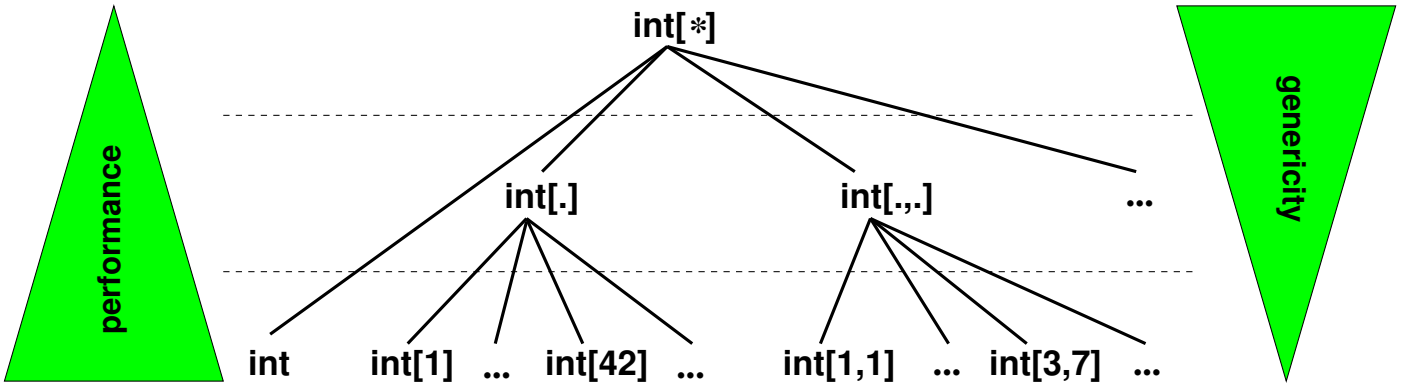
dim: 0
shape: [ ]
data: [42]

Shapely Array Type Hierarchy



AUD : Array of Unknown Dimension
AKD : Array of Known Dimension
AKS : Array of Known Shape

Genericity vs Performance Trade-Off



What software engineering principles demand:

- generic programs
- wide-spread, uniform applicability
- abstraction and composition

What the machine needs for performance:

- code customised to processed data
- exploit compile time knowledge for optimisation
- overcome abstraction boundaries

Why is generic code less efficient ?

Runtime representation of data:

- Rank and or shape vector need to be maintained dynamically
- More memory management, more function parameters, etc.

Why is generic code less efficient ?

Runtime representation of data:

- Rank and or shape vector need to be maintained dynamically
- More memory management, more function parameters, etc.

Runtime representation of code:

- AKS+AKD: static loop nestings
- AUD: complicated code to mimick multi-dimensional space

Why is generic code less efficient ?

Runtime representation of data:

- Rank and or shape vector need to be maintained dynamically
- More memory management, more function parameters, etc.

Runtime representation of code:

- AKS+AKD: static loop nestings
- AUD: complicated code to mimick multi-dimensional space

Shape compatibility / Boundary violations:

- Static guarantees vs dynamic checks

Why is generic code less efficient ?

Runtime representation of data:

- Rank and or shape vector need to be maintained dynamically
- More memory management, more function parameters, etc.

Runtime representation of code:

- AKS+AKD: static loop nestings
- AUD: complicated code to mimick multi-dimensional space

Shape compatibility / Boundary violations:

- Static guarantees vs dynamic checks

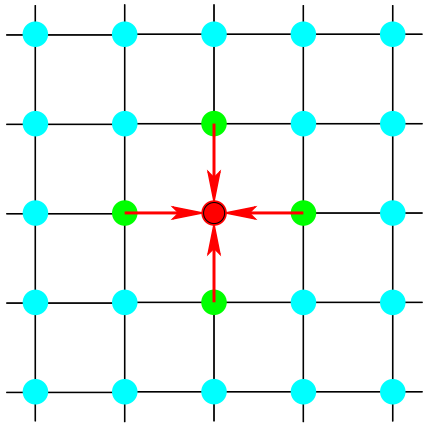
Effectiveness of optimisation:

- SAC promotes a (very) compositional programming style
- Despite C-like syntax: abstraction, abstraction, abstraction
- For performance: resolve abstractions through optimisation

Case Study: Convolution with Cyclic Boundary Conditions

Algorithmic principle:

Iteratively compute the weighted sums of neighbouring elements with cyclic neighbourhoods and dynamic convergence check



Implementation in SAC:

```
double[*] convolution (double[*] A, double eps)
{
  do {
    A_old = A;
    A = convolution_step( A_old);
  }
  while (!is_convergent( A, A_old, eps));

  return A;
}
```

Convolution in SAC

Convergence criterion:

```
bool
is_convergent (double[*] new, double[*] old, double eps)
{
  return all( abs( new - old) < eps);
}
```

Convolution step:

```
double[*] convolution_step (double[*] A)
{
  R = A;
  for (i=0; i<dim(A); i++) {
    R = R + rotate( i, 1, A) + rotate( i, -1, A);
  }
  return R / tod( 2 * dim(A) + 1);
}
```

Rotation from the SAC Standard Library

```
double[+] rotate (int dimension, int count, double[+] A)
{
  if (count == 0) {
    result = A;
  }
  else {
    max_rotate = shape(A)[dimension];
    if( max_rotate == 0) {
      result = A;
    }
    else {
      count = count % max_rotate;
      if (count < 0) {
        count = count + max_rotate;
      }

      offset = 0 * shape(A);
      offset[dimension] = count;

      slice_shp = shape(A);
      slice_shp[dimension] = count;

      result = with {
                 ( offset <= iv <= . ) : A[iv-offset];
               } : modarray( A );

      result = with {
                 ( . <= iv < slice_shp ) : A[shape(A)-slice_shp+iv];
               } : modarray( result );
    }
  }
  return result;
}
```

How Can We Reconcile Genericity and Performance ?

Observation:

- Often small numbers of different shapes prevail.
- Specialisation !!
- Large-scale static optimisation !!

How Can We Reconcile Genericity and Performance ?

Observation:

- Often small numbers of different shapes prevail.
- Specialisation !!
- Large-scale static optimisation !!

Limitations:

- Code obfuscation
- Arrays obtained from external source (e.g. file)
- Functions called from external environment (e.g. C code)

Solution: Adaptive Runtime Specialisation

Observations:

- Often small numbers of different shapes prevail
- All shapes known at runtime
- Multiple compute cores available
- Even with linear speedups one or two cores less hardly matters

Solution: Adaptive Runtime Specialisation

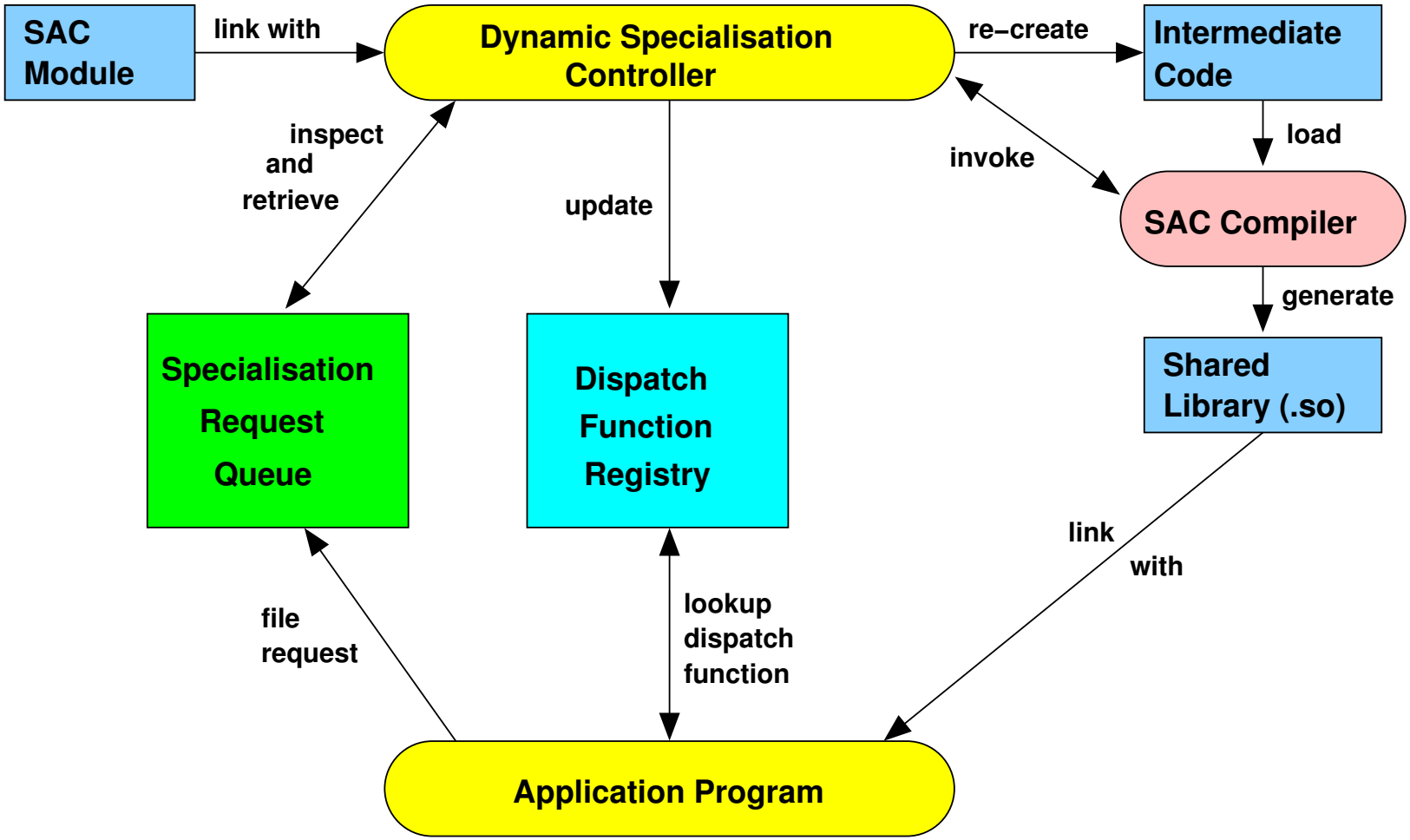
Observations:

- Often small numbers of different shapes prevail
- All shapes known at runtime
- Multiple compute cores available
- Even with linear speedups one or two cores less hardly matters

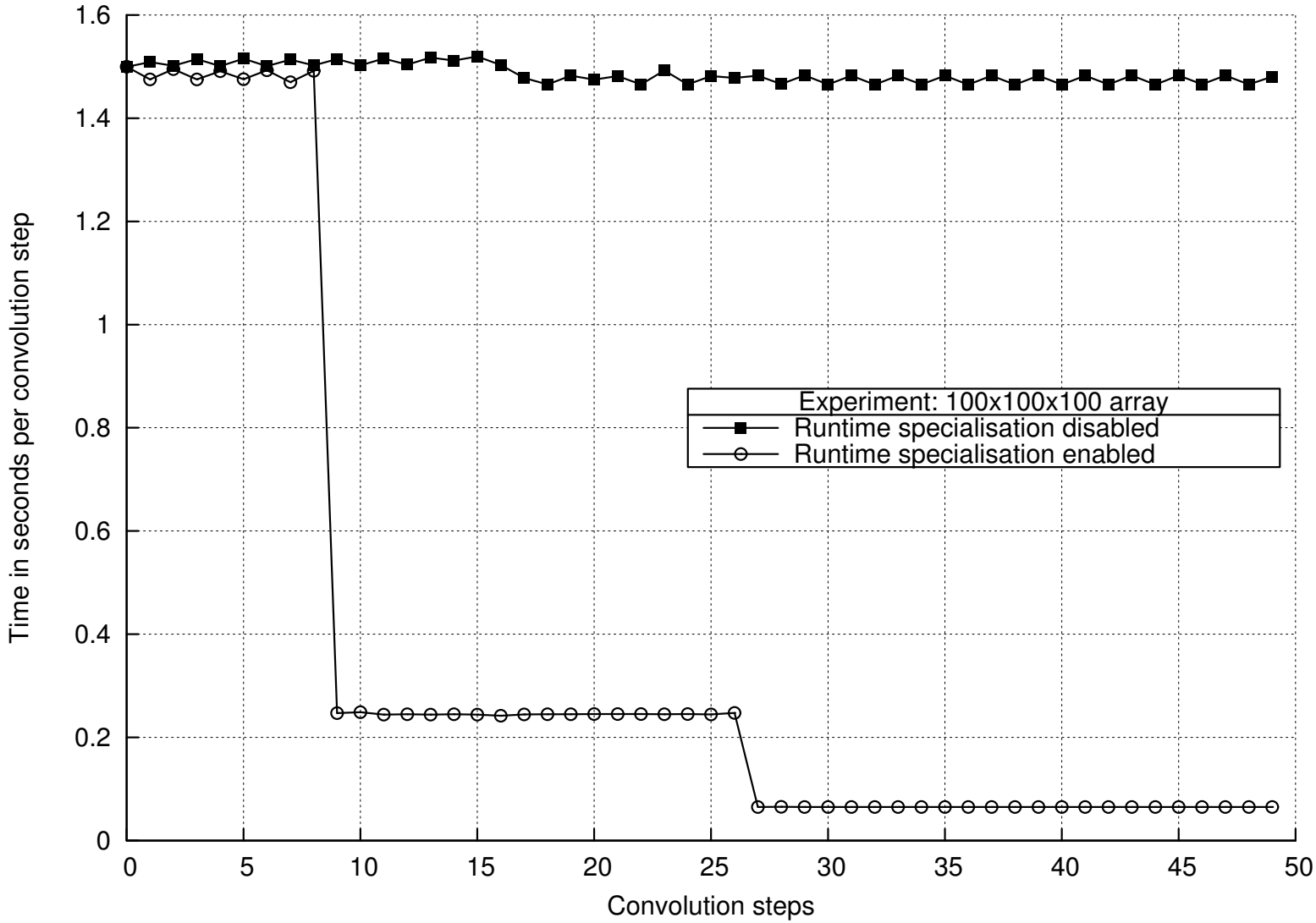
Idea:

- Set one core aside for dynamic code adaptation
- Incrementally generate more efficient code as shape information becomes available
- Accumulate adapted code in running process through dynamic linking
- Use adapted code as soon as available through dynamic dispatch

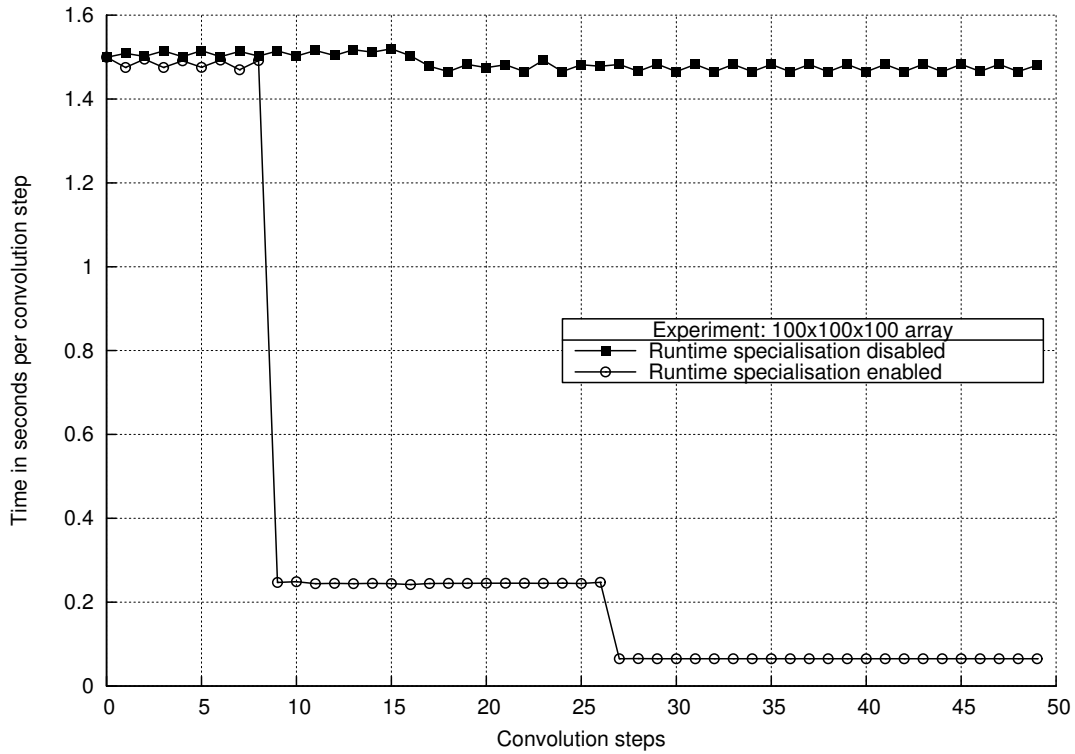
Adaptive Dynamic Compilation Architecture



Evaluation Example: 3-d Convolution 100x100x100



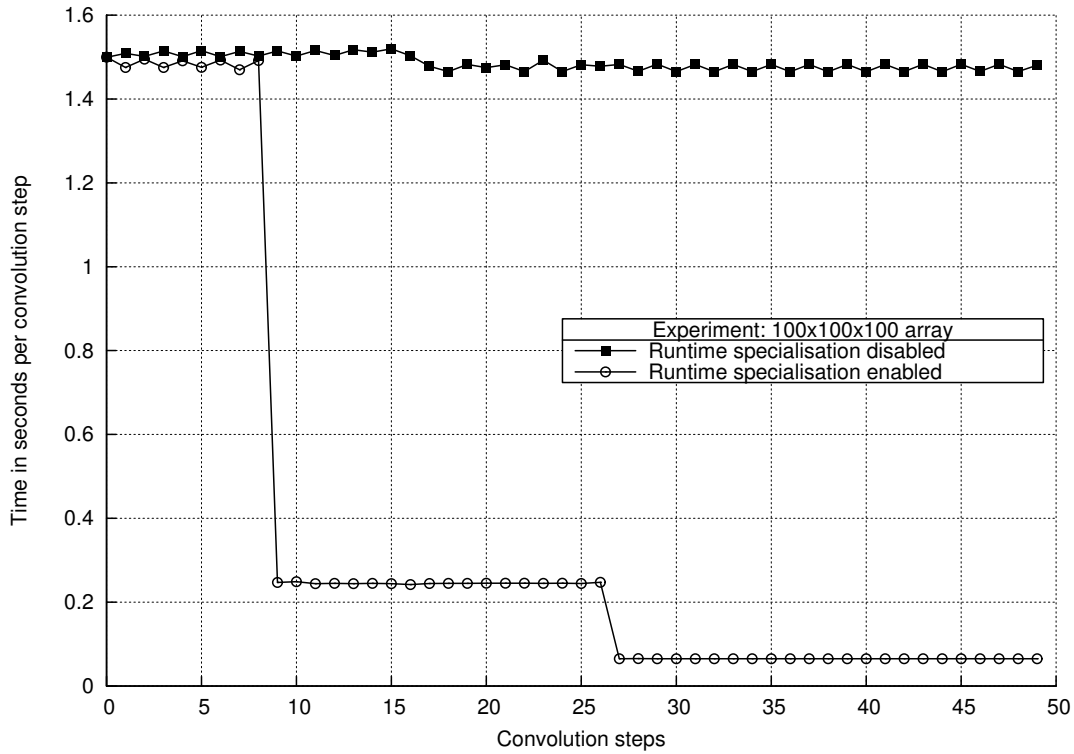
Evaluation Example: 3-d Convolution 100x100x100



Key question:

How can we speed up the availability of adapted code ?

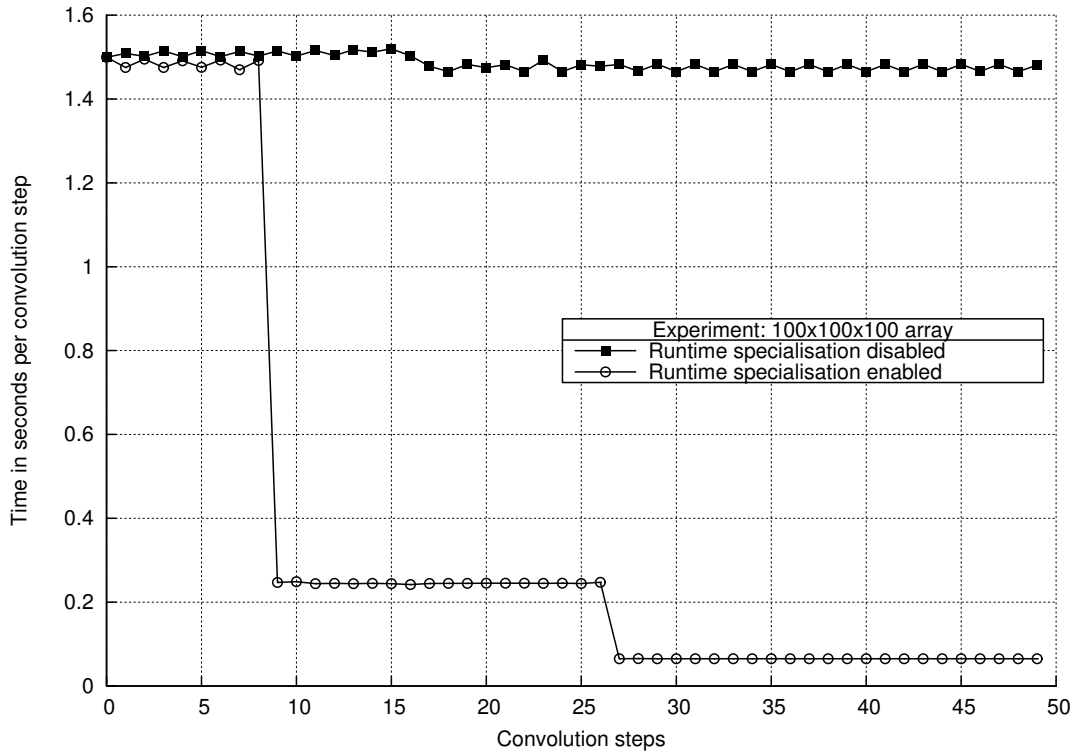
Evaluation Example: 3-d Convolution 100x100x100



Key question:
How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?

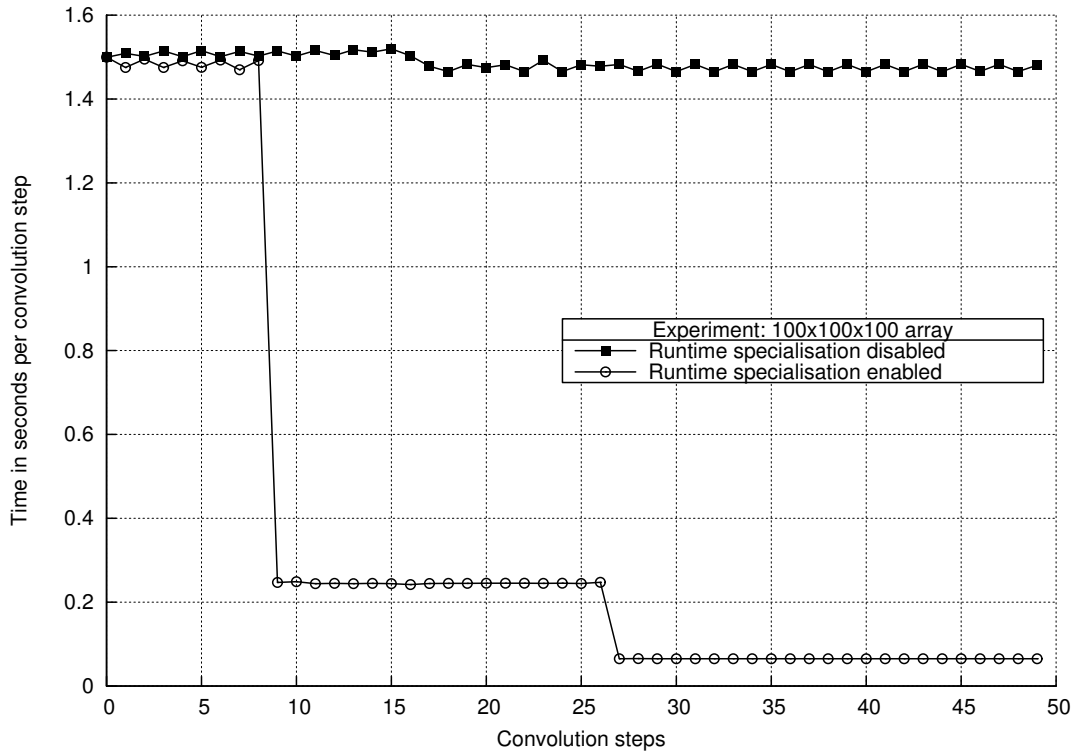
Evaluation Example: 3-d Convolution 100x100x100



Key question:
How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?
- Parallelise code adaptation !

Evaluation Example: 3-d Convolution 100x100x100

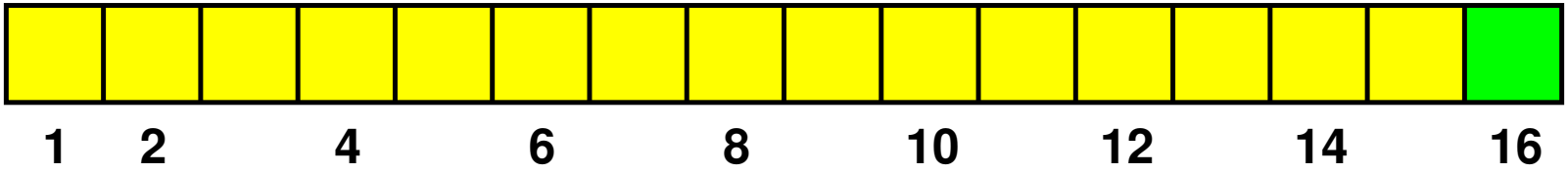


Key question:
How can we speed up the availability of adapted code ?

- Accelerate SAC compiler ?
- Parallelise code adaptation !
- Make adapted code persistent !

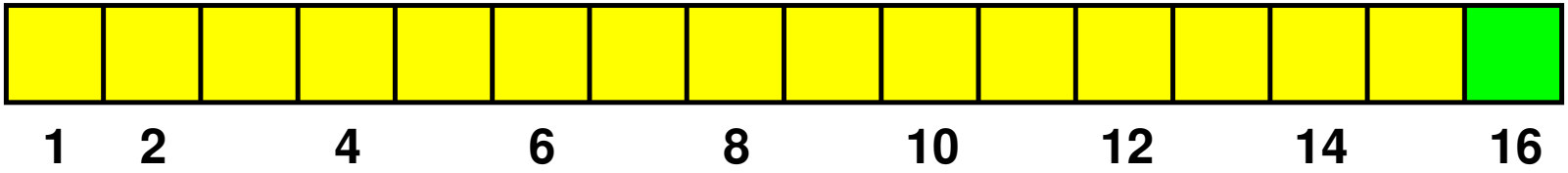
Parallel Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:



Parallel Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:

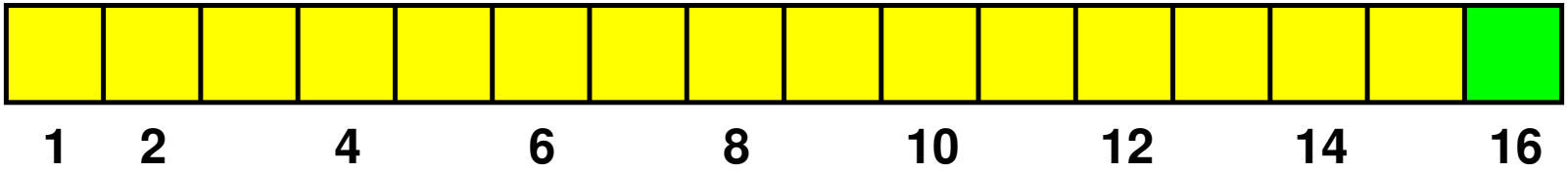


Key observations:

- Dynamic specialisations are time-consuming
- Adapted functions only become available with delay
- Insight: One specialisation core only is suboptimal

Parallel Asynchronous Adaptive Specialisation

First approach: dedicated specialisation core:

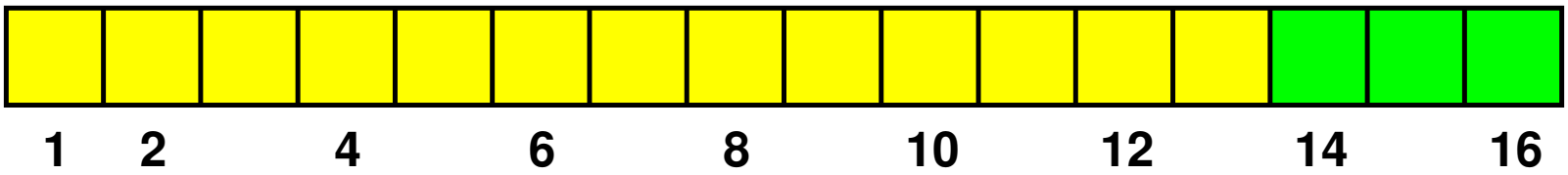


Key observations:

- Dynamic specialisations are time-consuming
- Adapted functions only become available with delay
- Insight: One specialisation core only is suboptimal

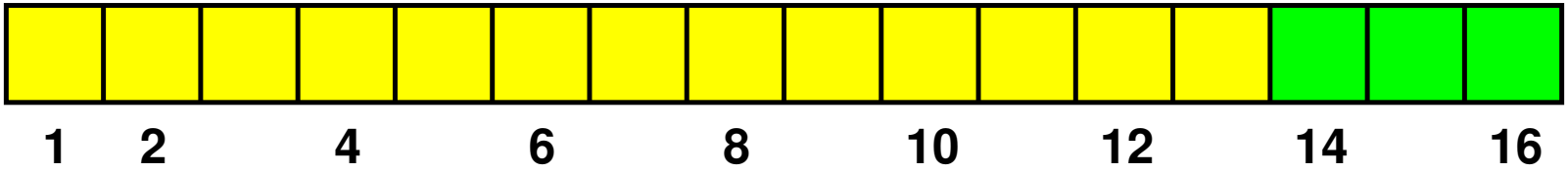
Solution:

- Use configurable division of cores:



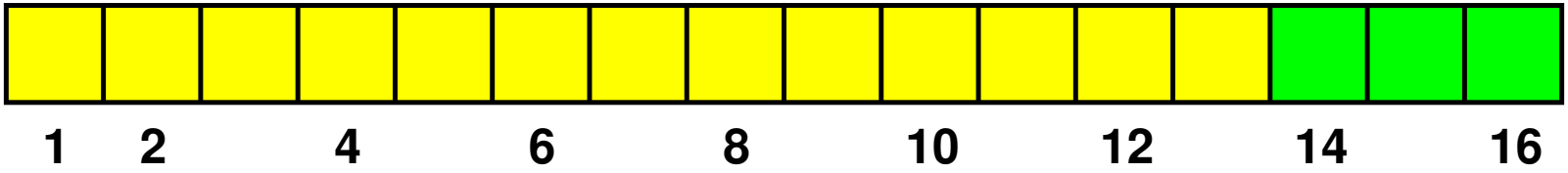
Concurrent Dynamic Specialisation Reloaded

Second approach: configurable number of specialisation core



Concurrent Dynamic Specialisation Reloaded

Second approach: configurable number of specialisation core

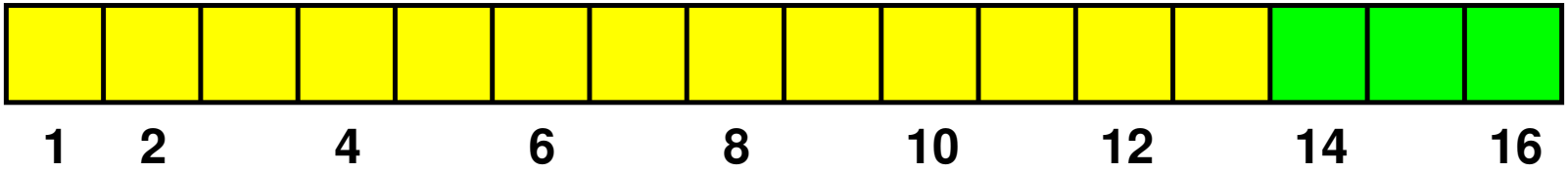


More observations:

- Dynamic specialisation often reaches fixed point
- Specialisations cores idle after a while
- Insight: Any fixed number of specialisation cores is suboptimal

Concurrent Dynamic Specialisation Reloaded

Second approach: configurable number of specialisation core

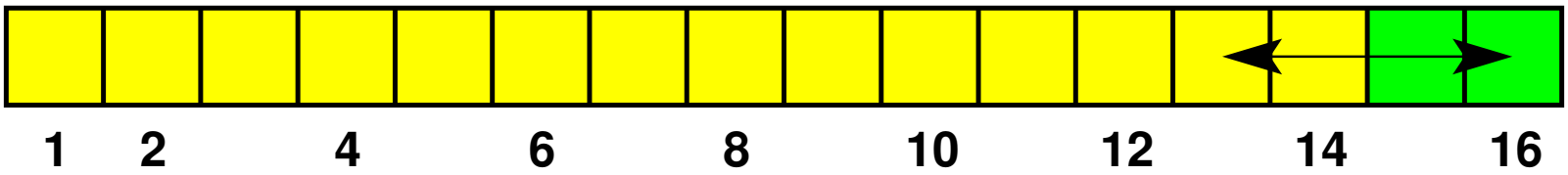


More observations:

- Dynamic specialisation often reaches fixed point
- Specialisations cores idle after a while
- Insight: Any fixed number of specialisation cores is suboptimal

Solution:

- Dynamically adjust number of specialisation cores:



Persistent Dynamic Adaptation

More observations:

- Dynamic code adaptation is for one program run
- Insight: the very same dynamic specialisations are built again and again

Persistent Dynamic Adaptation

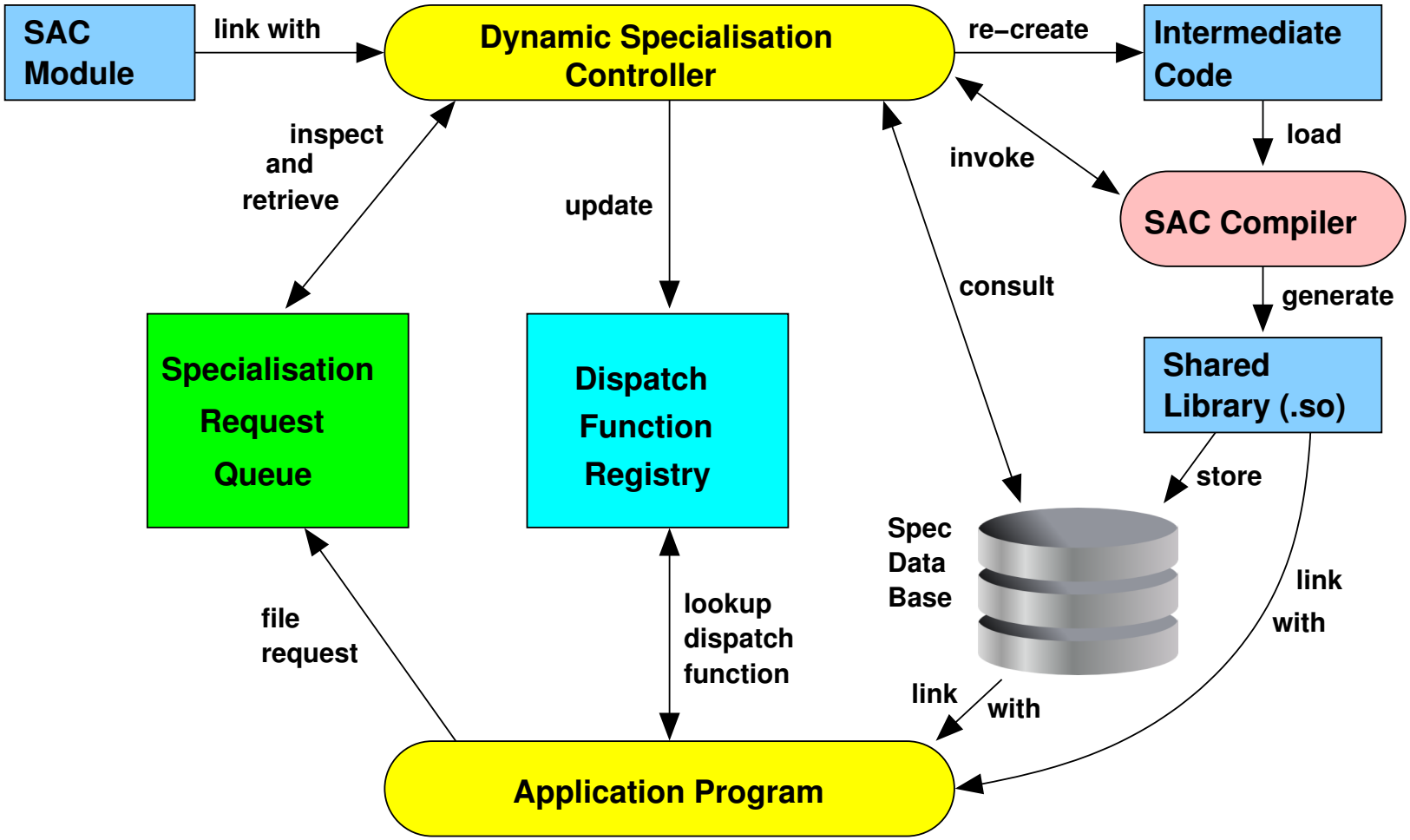
More observations:

- Dynamic code adaptation is for one program run
- Insight: the very same dynamic specialisations are built again and again

Solution:

- Store dynamic specialisations in installation-wide persistent storage
- Incrementally update the binary format of a module with new specialisations as they materialise
- Use replacement policies as in cache memories (e.g. least recently used)
- Learn which shapes are relevant.

Adaptive Dynamic Compilation Architecture 2.0



Conclusion

Classical trade-off:

Software Engineering Principles

vs

Runtime Performance Requirements

Solution:

Large-scale Specialisation and Optimisation

plus

Dynamic Code Adaptation

Future work:

- Complete implementation
- More case studies
- Extensive evaluation