CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2012; **24**:499–516
Published online 21 September 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1842

SPECIAL ISSUE PAPER

# Asynchronous adaptive optimisation for generic data-parallel array programming

Clemens Grelck[1,*,†], Tim van Deurzen[1], Stephan Herhut[2] and Sven-Bodo Scholz[2]

[1]*Institute of Informatics, University of Amsterdam, 1098 XH Amsterdam, The Netherlands*
[2]*School of Computer Science, University of Hertfordshire, AL10 9AB Hatfield, UK*

## SUMMARY

Programming productivity very much depends on the availability of basic building blocks that can be reused for a wide range of application scenarios and the ability to define rich abstraction hierarchies. Driven by the aim for increased reuse, such basic building blocks tend to become more and more generic in their specification; structural as well as behavioural properties are turned into parameters that are passed on to lower layers of abstraction where eventually a differentiation is being made. In the context of array programming, such properties are typically array ranks (number of axes/dimensions) and array shapes (number of elements along each axis/dimension). This allows for abstract definitions of operations such as element-wise additions, concatenations, rotations, and so on, which jointly enable a very high-level compositional style of programming, similar to, for instance, MATLAB. However, such a generic programming style generally comes at a price in terms of runtime overheads when compared against tailor-made low-level implementations. Additional layers of abstraction as well as the lack of hard-coded structural properties often inhibits optimisations that are obvious otherwise. Although complex static compiler analyses and transformations such as partial evaluations can ameliorate the situation to quite some extent, there are cases, where the required level of information is not available until runtime. In this paper, we propose to shift part of the optimisation process into the runtime of applications. Triggered by some runtime observation, the compiler asynchronously applies partial evaluation techniques to frequently used program parts and dynamically replaces initial program fragments by more specialised ones through dynamic re-linking. In contrast to many existing approaches, we suggest this optimisation to be done in a rather non-intrusive, decoupled way. We use a full-fledged compiler that is run on a separate core. This measure enables us to run the compiler on its highest optimisation-level, which requires non-negligible compilation times for our optimisations. We use the compiler's type system to identify the potential dynamic optimisations. And we use the host language's module system as a facilitator for the dynamic code modifications. We present the architecture and implementation of an adaptive compilation framework for Single Assignment C, a data-parallel array programming language. Single Assignment C advocates shape-generic and rank-generic programming with arrays. A sophisticated, highly optimising compiler technology nevertheless achieves competitive runtime performance. We demonstrate the suitability of our approach to achieve consistently high performance independent of the static availability of array properties by means of several experiments based on a highly generic formulation of rank-invariant convolution as a case study. Copyright © 2011 John Wiley & Sons, Ltd.

Received 16 February 2011; Revised 18 June 2011; Accepted 28 July 2011

KEY WORDS: adaptive compilation; data-parallel computation; shape generic programming

*Correspondence to: Clemens Grelck, Institute of Informatics, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands.
†E-mail: c.grelck@uva.nl

## 1. INTRODUCTION

Software engineering is characterised by a fundamental dilemma: programming productivity versus runtime performance. In an ideal world, program code would be highly generic, easy to understand, abstract, well maintainable and portable, while at the same time, it would harness the highest levels of performance that the hardware it runs on is able to deliver. In the real world, any software is somewhere in the continuous space between these extremes. Different software engineering communities make different choices where to locate themselves in this continuous space depending on external demands. If time-to-market is essential for success while the resulting program only needs to run 'fast enough', the choice will certainly be a different one than in a situation where performance is essential as in high-performance computing. This motivates the use of different programming languages and tool chains in different environments.

Optimising compilers aim at bridging the gap between the programmer's desire for generic, re-usable programs adhering to software engineering principles such as abstraction and composition and the necessities of executable code to achieve high runtime performance in sequential and, increasingly important, (implicitly) parallel execution. Optimising compilers, analyse program code and infer static properties that trigger program transformations as appropriate. In essence, they try to overcome the general dilemma; with varying success, of course.

The effectiveness of static analysis often is limited for various reasons. It may depend on finding a suitable order of program transformations; it may even require transformations that introduce overheads, so that they can enable further optimisations. To make matters worse, most analytical approaches often fail to identify the best possible optimisation strategies analytically. Recent research in this area suggests that purely analytical approaches are less successful than those approaches based on machine learning techniques [1].

Even if these technical challenges could be overcome completely, there inevitably remains the problem that some crucial information about the data to be processed may simply not be available up until runtime. In particular, in the context of generic array programming, lack of structural information for input data, or intermediate results can have a detrimental impact on program runtimes. The efficient execution of applications written in a high-level combinator style requires, for instance, advanced loop fusion techniques [2]. Their effectiveness, however, crucially depends on the level of static knowledge about the data involved. Even advanced symbolic analyses [3, 4], although often successful, cannot fully eliminate this dependency. Ultimately, only radically specializing generic programs to actual input data guarantees sufficient static knowledge for a compiler to produce highly efficient code. Yet, for many applications, the increased size of executables and additional compilation time caused by the specialisation for large numbers of potential inputs is prohibitive.

Driven by this observation, we propose to use a more dynamic approach. Instead of producing a range of specialized versions at compile time, we delay program specialization until runtime. Other than common approaches to runtime specialisation, we make use of an existing full-fledged compiler without requiring any substantial modifications to it. The key ideas are to run the compiler asynchronously on a separate core, and to employ existing mechanisms in the compiler to trigger and to implement the actual code modifications. We demonstrate the feasibility of our approach in the context of the compilation framework for the data-parallel functional array language, Single Assignment C (SAC) [5]. SAC advocates shape-generic and rank-generic programming on multi-dimensional arrays: SAC supports functions that abstract from the concrete shapes (extent along dimensions) and even from the concrete ranks (number of dimensions) of argument arrays. Furthermore, functions yield result arrays whose shape and rank are determined by some computation in the function itself. Depending on the amount of compile time structural information, the type system of SAC distinguishes three classes of arrays that induce three different runtime array representations: From non-generic arrays whose structure is fully known at compile time, via shape-generic arrays where only the rank but not the extent of each dimension is known up to rank-generic arrays that have a fully dynamic shape.

From a software engineering point of view, it is (usually) desirable to specify functions on rank-generic input type(s) to maximise opportunities for code reuse. Typical examples for such rank-generic operations are extensions of scalar operators (arithmetic, logical, relational etc.) to entire arrays in an element-wise way or common structural operations like shifting and rotation along one or multiple axes of an array. In fact, rank-generic functions prevail in the extensive SAC standard library.

The benefits that this genericity offers come at a price. In comparison with non-generic code, the runtime performance of equivalent operations is substantially lower for shape-generic code and, again, substantially lower for rank-generic code [6]. The reasons are manifold and their individual impact operation-specific, but three categories can be identified notwithstanding: First, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, for example no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Second, many of the SAC-compiler's advanced optimisations [7, 8] are just not as effective for generic code because the necessary code properties to trigger certain program transformations cannot be inferred. Third, in automatically parallelised code [9], many organisational decisions must be postponed until runtime, and the ineffectiveness of optimisation leads to excessive numbers of synchronisation barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC-compiler aggressively specialises rank-generic code into shape-generic code, and shape-generic code into non-generic code. However, regardless of the effort put into compiler analyses for rank and shape specialisation, this approach is fruitless if the necessary rank and shape information is simply not available at compile time for whatever reason. Data may be read from a file at runtime, or SAC code is called externally from a non-SAC environment via the sac4c foreign language interface [10]. In particular, the latter is more and more common as we use SAC in conjunction with the component-based coordination language S-Net [11].

To mitigate the negative effect of generic code on runtime performance where specialisation is not an option for one or more of the aforementioned reasons, we propose an adaptive compilation framework that incrementally adapts shape-generic and rank-generic code to the concrete shapes and ranks used in a specific instance of a program. After all, at runtime, full shape information is always available. Our approach is motivated by the observation that the number of different array shapes that effectively appear in generic array code, although theoretically unbounded, often is relatively small in practice.

Two aspects set our adaptive compilation framework apart from existing just-in-time compilation and dynamic optimisation/code tuning: For one, we dynamically adapt generic code to structural properties of the data it operates on, whereas just-in-time compilation of byte code (or similar) aims at adapting code to the execution environment, for example by generating native machine code. The second and probably more far-reaching difference is that we inherently assume a multicore execution environment where computing resources are available in abundance and often cannot completely be exploited by a running program in an efficient way. Although the SAC-compiler is equipped with very effective implicit parallelisation technology [9], experience says that the difference between using 14 cores of a 16-core machine and using all cores for running a given program is often marginal because the additional overhead for organising parallel execution more and more outweighs the benefit with each core joining in into collaborative execution. At this point, we propose to set apart a small (configurable) number of cores for the purpose of incrementally adapting the binary code base to the array shapes actually appearing during a program run. Our approach takes dynamic recompilation out of the critical path of an application. This property is instrumental in using a heavy-weight, highly optimising compiler like `sac2c` in an online setting.

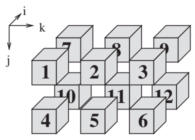
The remainder of the paper is organised as follows. Section 2 provides a few more details on the design of SAC. In Section 3, we demonstrate the generic programming style of SAC by means of a small case study, generic convolution. In Section 4, we sketch out the compilation tool chain of SAC and explain why generic and non-generic codes may have very different runtime behaviour. We present our ideas on adaptive compilation in more detail in Section 5 and discuss implementation issues in Section 6. In Section 7, we report on extensive experiments applying our adaptive compilation framework to the case study code. Eventually, we browse through related work in Section 8, draw conclusions in Section 9 and sketch out directions of future work in Section 10.

## 2. SINGLE ASSIGNMENT C IN A NUTSHELL

As the name ‘Single Assignment C’ suggests, SAC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate familiarisation for programmers who rather have a background in imperative languages than in declarative languages. Core SAC is a functional, side-effect free subset of C: we interpret assignment sequences as nested let-expressions, branching constructs as conditional expressions and loops as syntactic sugar for tail-end recursive functions. Details on the design of SAC and the functional interpretation of imperative-looking code can be found in [5]. Despite the radically different underlying execution model (context-free substitution of expressions versus step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SAC code whereas the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation [9].

On top of this language kernel, SAC provides genuine support for truly multidimensional and truly stateless/functional arrays advocating a shape-generic and rank-generic style of programming. Conceptually, any SAC expression denotes an array; arrays can be passed to and from functions call-by-value. A multidimensional array in SAC is represented by a *rank scalar* defining the length of the *shape vector*. The elements of the shape vector define the extent of the array along each dimension, and the product of its elements defines the length of the *data vector*. The data vector contains the array elements (in row-major order). Figure 1 shows a few examples for illustration. Notably, the underlying array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Furthermore, we achieve a complete separation between data assembled in an array and the structural information (rank and shape).

The type system of SAC (at the moment) is monomorphic in the element type of an array but polymorphic in the structure of arrays, that is rank and shape. As illustrated in Figure 2, each element type induces a conceptually unbounded number of array types with varying static structural restrictions on arrays. These array types essentially form a hierarchy with three levels. On the lowest level, we find non-generic types that define arrays of fixed shape, for example `int[3,7]` or just `int`. On an intermediate level of genericity, we find arrays of fixed rank, for example `int[.,.]`. And, on the top of the hierarchy, we find arrays of any rank, for example `int[*]`. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.



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

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

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

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

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

42

rank: 0
shape: [ ]
data: [42]

Figure 1. Truly multidimensional arrays in SAC and their representation by data vector, shape vector and rank scalar.

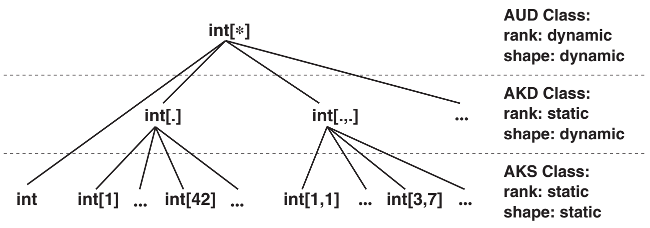


Figure 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS).

SAC only provides a small set of built-in array operations. Essentially, there are primitives to retrieve data pertaining to the structure and contents of arrays, for example an array's rank (dim(*array*)) or its shape(shape(*array*)). A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*[*idxvec*]. The use of a vector for the purpose of indexing into an array is crucial in a rank-generic setting: if the number of dimensions of an array is left unknown at compile time, any syntax that uses a fixed number of indices (e.g. comma-separated) makes no sense whatsoever.

Whereas simple (one-dimensional) vectors can be written as shown in Figure 1, that is as a comma-separated list of expressions enclosed in square brackets, any rank-generic or shape-generic array is defined by means of WITH-loop expressions. The WITH-loop is a versatile SAC-specific array comprehension or map-reduce construct. Because the ins and outs of WITH-loops are not essential to know for reading the rest of the paper, we skip any detailed explanation here and refer the interested reader to [5] for a complete account.

## 3. CASE STUDY: GENERIC CONVOLUTION

In order to illustrate the shape-generic high-level programming style typical of SAC and, in consequence, to demonstrate the effect of the proposed adaptive compilation framework, we introduce a small case study: generic convolution with iteration count and convergence check. This computationally non-trivial and application-wise highly relevant numerical kernel fits on half a page of SAC code; Figure 3 contains the complete implementation.

```
module Convolution;

use Array: all;

export {convolution};

double[*] convolution (double[*] A, double epsilon, int max_iterations)
{
  i = 0;

  A_new = A;

  do {
    A_old = A_new;
    A_new = convolution_step( A_old);
    i += 1;
  }
  while (!is_convergent( A_new, A_old, epsilon) && i < max_iterations);

  return ( A_new);
}

inline double[*] convolution_step (double[*] A)
{
  R = A;

  for (i=0; i<dim(A); i++) {
    R += rotate( i, 1, A) + rotate( i, -1, A);
  }

  return ( R / tod( 2 * dim(A) + 1));
}

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

Figure 3. Case study: generic convolution kernel with convergence check.

The first line of code defines a module convolution. This module makes intensive use of the array module from the SAC standard library that defines a large number of typical array operations such as array extensions of the usual scalar primitive operators and functions, structural operations like shifting and rotation or reduction operations like sum or product of array elements. The last line of the module header declares that our module convolution exports a single symbol, that is the function convolution.

The function convolution is defined in lines 7–21; it expects three arguments: an array A of double precision floating point numbers of any shape and any rank (numbers of dimensions), which is the array to be convolved, a double precision floating point number epsilon that defines the desired level on convergence and, last but not least, an integer number max iterations that is supposed to prematurely terminate the convolution after a given number of iterations regardless of the convergence behaviour.

The body of the function convolution essentially consists of the iteration loop, a C-style **do/while**-loop. This nicely demonstrates the close syntactical relationship between SAC and C. Semantically, however, the SAC **do/while**-loop is merely syntactic sugar for an inlined tail-recursive function. Nonetheless, the SAC programmer hardly needs to reason about such subtle semantical differences as the observable runtime behaviour of the SAC code is exactly the same as one would expect from the corresponding C code.

Within the **do/while**-loop of function convolution, we essentially perform a single convolution step that itself is implemented by the function convolution step defined in lines 23–32. This function expects an array A of double precision floating point numbers of any shape and any rank, and yields a new array of the same shape and rank. In our example, we chose cyclic boundary conditions as exemplified by the use of the rotate function from the SAC standard array library. In fact, the function rotate(index, offset, array) creates an array that has the same rank and shape as the argument array, but with all elements rotated by offset index positions along array axis (or dimension) index. With the **for**-loop in lines 27–29, we rotate the argument array twice in each dimension, by one element towards decreasing and by one element towards increasing indices. Each time, we combine the rotated arrays using element-wise addition, as implemented by an overloaded version of the + operator. In essence, this implements a rank-invariant direct-neighbour stencil operation, that is in the one-dimensional case, we have a three-point stencil, in the two-dimensional case, a five-point stencil, in the three-dimensional case, a seven-point stencil and so on.

In many concrete applications, we will have different weights for different neighbours. In order to bound the complexity of our case study, we refrain from supporting this here, albeit such an extension would be rather straightforward. Instead, we merely compute the arithmetic mean, that is all neighbours and the old value have the same weight. To achieve this, we divide all elements of array R by the number of neighbours, plus one for the old value. The function `tod` merely converts an integer number into a value of type **double**.

Coming back to the definition of the function convolution, we may want to have a closer look at the loop predicate of the **do/while**-loop in line 18. We continue as long as we neither detect convergence nor the maximum number of iterations is reached. Whereas the latter requires a simple comparison on integer scalar values, the former makes use of the generic convergence test defined in lines 34–37. The function convergent checks whether for all elements of the argument arrays new and old, the absolute value of the difference is less than the given convergence threshold epsilon. This function definition is a nice example of the SAC programming methodology that advocates the implementation of new array operations by composition of existing ones. All four basic array operations used here, that is element-wise subtraction, element-wise absolute value, element-wise comparison with a scalar value and reduction with Boolean conjunction (all), are defined in the SAC standard library.

We have already discussed potential further generalisation of this convolution kernel with respect to using different weights for different neighbour elements. Another potential extension would be a generalisation of the stencil itself to cover different notions of neighbourship in a rank-invariant specification. Again, we refrain from doing so here, as it would make the code more difficult to understand, while we do not expect any different behaviour with respect to the adaptive compilation framework we propose in this paper.

## 4. SINGLE ASSIGNMENT C COMPILATION TOOL CHAIN

Figure 4 shows the essential components of the SAC compiler tool chain. Following the inevitable lexical and syntactic analysis, we first do a *functionalisation* of the intermediate code. In this compiler phase, most of the imperative-looking features of SAC like for instance C-style branches and loops are converted into properly functional conditional expressions and fully-fledged tail-recursive functions, respectively. The following type inference and specialisation compiler phase infers as concrete as possible array types based on static analysis of the code. The by far largest and most important part of the SAC compiler tool chain, however, are the high-level optimisations. This is the heart of the compiler. We assemble a large number of compiler optimisations and aim at computing the fixed point of intermediate program representation with respect to the various program transformations. Among them are many text book compiler optimisations, such as function inlining or common subexpression elimination, but likewise a large number of SAC-specific optimisations geared towards the generic array programming context.

Following an aggressive, typically large-scale reorganisation of intermediate code by the various compiler optimisations, we have two fundamental lowering steps. The memory management compiler phase introduces symbolic memory allocation and de-allocation as well as reference counting operations into the code [12]. The *defunctionalisation* compiler phase revokes many of the transformations made earlier; for instance, tail-recursive functions are again transformed into loops. The resulting intermediate code may, on demand, automatically be parallelised [9] before we finally generate ANSI C code. Any ANSI-compliant C compiler can then be used to obtain executable binary code.

We highlight the architecture of the SAC compiler tool chain here to illustrate how little it resembles a lightweight just-in-time compiler. Instead, it is a heavy-weight highly optimising compiler and designed for exactly this purpose: generating efficient code, not efficiently generating code. Of course, one could accelerate compilation by reducing the amount of optimisation or even switching off optimisation entirely, but that runs counter our purpose. We particularly aim at exploiting the optimisation capabilities with the proposed adaptive compilation framework, just at application runtime.

In the remainder of this section, we will elaborate on a few issues mentioned in the introduction that explain a potentially drastic performance difference between generic array code and code

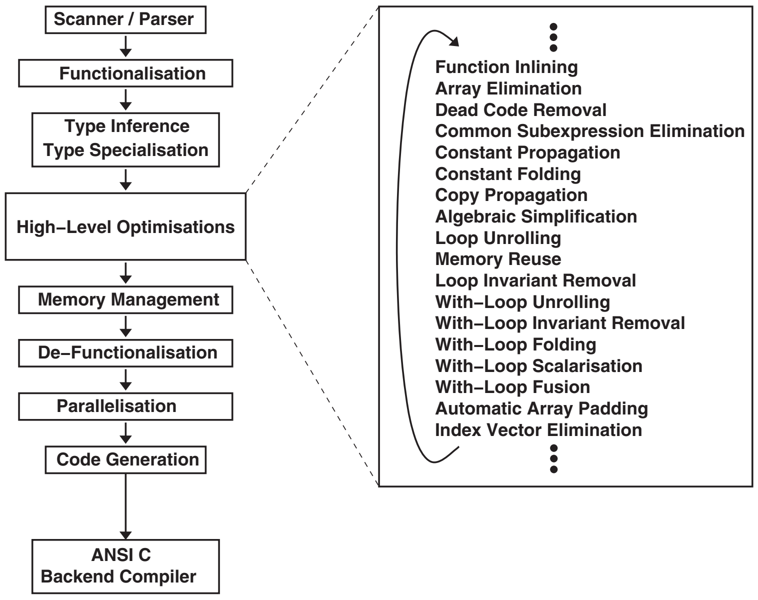


Figure 4. Organisation of the SAC compiler tool chain.

```
      float  *X_data;

const int    X_dim =2;
const int    X_shp0=42;
const int    X_shp1=21;
```

```
      float  *X_data;
      int  *X_desc;
const int    X_dim =2;
      int    X_shp0=X_desc[1];
      int    X_shp1=X_desc[2];
```

```
float  *X_data;
int  *X_desc;
int    X_dim=X_desc[0];
```

Figure 5. Different runtime representations for arrays of (statically) known shape (AKD) left, arrays of known dimension (AKD) center and arrays of unknown dimension (AUD) right.

that is either originally non-generic or specialised by the compiler at an early compilation stage. Going from bottom to top, Figure 5 illustrates the different runtime representations used by SAC for arrays of (statically) known shape (AKS), arrays of known dimension (AKD) and arrays of unknown dimension (AUD) for the example of a $42 \times 21$-matrix of single precision floating point numbers X. In each case, we can identify the data vector X_data. In the AKS case, we can store all structural information as constants, that is the rank of the array (X_dim) and the extent along the two dimensions (X_shp0 and X_shp1). Whenever structural information of the matrix X is needed further in the code, the C compiler can immediately use these constant values in assembly generation. In the AKD case, the data vector X data must be accompanied by a *descriptor* X desc that dynamically carries the structural properties of an array together with the data vector around between function invocations. Still, the rank can be stored as a constant, and the shape information can be cached in registers within a function context for more efficient access. Last but not least, in the AUD case, we have the same descriptor as in the AKD case, but here we also need to extract the rank from the descriptor. We may cache the rank information in a register, but because we do not know the number of dimensions at compile time, we cannot do the same for the shape information. Consequently, any access to this information inflicts a costly memory load operation to retrieve the information directly from the descriptor.

In analogy to the different data representations shown in Figure 5, we can also identify an important difference in the generated code operating on such data. As long as we at least know the rank of arrays statically, (irregular) operations on them can still be represented as efficient nestings of **for**-loops. As soon as we are confronted with rank-generic code, we need to mimic a multi-dimensional loop structure by a single loop and costly retrieval of a conceptually multi-dimensional index location from a single scalar index.

The third major source of overhead of generic code as opposed to non-generic code stems from reduced effectiveness of high-level optimisation. As an example, consider our case study code in Figure 3. If we know the rank of the argument array to function convolution_step at compile time (e.g. through specialisation), we can unroll the **for**-loop in lines 27–29 as typically the rank will be a relatively small integer number. As a consequence, the induction variable i in the application of rotate becomes a constant. In conjunction with the already constant rotation offset, we can highly optimise/adapt the rotation operation to the concrete requirement. Furthermore, we can condense the now statically known number of rotations into a single array comprehension (**with**-loop) from which we can generate executable code that closely resembles an optimised low-level imperative implementation of the convolution step.

## 5. ADAPTIVE COMPILATION FRAMEWORK

The architecture of our adaptive compilation framework is sketched out in Figure 6. A key design choice in our framework is the separation of the gathering of profiling information that triggers specialisation (bottom part of Figure 6) from the actual runtime specialisation itself (the grey boxes in the upper part of Figure 6). Our aim was to keep the implementation of the former as lean as possible to keep the impact on compiled application code minimal. Instead, most of the new functionality is encapsulated in the dynamic specialisation controller. The running program and the dynamic specialisation controllers communicate with each other exclusively via two shared data structures: the dispatch function registry and the specialisation request queue (shown as dark boxes in the center of Figure 6). This lean and well-defined interface facilitates the use of multiple specialisation controller

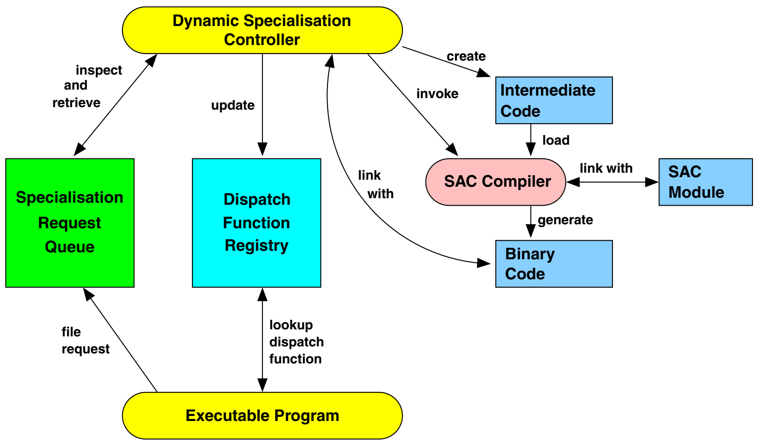


Figure 6. Architecture of our adaptive compilation framework.

instances on the one side and supports multithreaded execution of the program itself on the other side of the interface.

Our design makes use of the existing function call infrastructure within executable (binary) SAC programs generated by our SAC compiler `sac2c`. Such programs (generally) consist of binary versions of shape-specific, shape-generic and rank-generic functions. Any shape-generic or rank-generic function, however, is called indirectly through a *dispatch function* that selects the correct instance of the function to be executed in the presence of function overloading by the programmer and static function specialisation by the compiler. This dispatch function serves as an ideal hook to add further instances (specialisations) of functions created at runtime. Because adding more and more instances also affects function dispatch itself, we need to change the actual dispatch function each time we add further instances. To achieve this, we no longer call the dispatch function directly, but do this through a pointer indirection that allows us to exchange the dispatch function dynamically as needed. We call the central switchboard that implements the function call forwarding the *dispatch function registry*.

Figure 7 gives the pseudo code for function dispatch with dynamic specialisation. Before actually calling the dispatch function retrieved from the registry, we file a specialisation request in the *specialisation request queue*. Apart from information that allows us to uniquely identify the target of the call, that is, the function name, the module name where the function originates from, and so on, this request contains the concrete shapes of all actual arguments to the generic formal parameters of the called function. It is essential here that queuing specialisation requests is implemented as lightweight as possible as this operation is performed for every call to a generic function. To achieve this, we have slimmed the operation down as far as possible. Most information contained in the specialisation request is precomputed and preassembled at compile time. Furthermore, we do not perform any sanity checks during the enqueue operation. These are postponed until the item is later processed by a separate worker thread. This design is geared towards reducing

```
foo_dispatch( arguments ) {
    Collect rank and shape information from runtime descriptors of arguments.

    enqueue_request( "foo",
                     "moduleBar",
                     types,
                     shapes,
                     pointer into registry );

    foo_ptr = address stored in the registry;

    foo_ptr( arguments );
}
```

Figure 7. Pseudo code describing the interplay of function dispatch and runtime specialisation.

the impact of the proposed adaptive compilation framework on the genuine program execution to a minimum.

Within the same process that runs the *executable program*, one or more threads are set apart acting as *dynamic specialisation controllers*. A dynamic specialisation controller is in charge of the main part of the adaptive compilation infrastructure; it always runs asynchronous from the program itself. A dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the specialisation requested already exists or is currently in the process of being constructed. If so, the request is discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of the specialised function instance to be generated.

The dynamic specialisation controller then invokes the SAC-compiler `sac2c` on the intermediate representation, that is the dynamic specialisation controller effectively turns itself into the SAC-compiler. As such, it now starts the standard compilation process for the generated intermediate representation. During this process, the compiler dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SAC module system that was originally developed to support inter-module (compile time) optimisation [13].

Eventually, the SAC-compiler (with the help of a backend C compiler) generates another shared library containing binary versions of the specialised function(s) and one or more new dispatch functions taking the new specialisations into account in their decision. Following the completion of the SAC-compiler, the dynamic specialisation controller regains control.

Before attending to the next specialisation request, two tasks still need to be performed to enable the new specialised code in the running program. Firstly, the controller links the running process with the newly created shared library. As the module name chosen for the stub is unique, this will make a new symbol for the dispatch code of the specialised function available. In a second step, the controller updates the dispatch function registry with the new address of this new symbol for dispatch function(s) from the newly compiled library. As a consequence, any subsequent call to the now specialised function originating from the running program will directly be forwarded to the specialised instance rather than to the generic version and benefit from (potentially) substantially higher runtime performance without further overhead.

## 6. IMPLEMENTATION ASPECTS

We have extended the existing SAC compiler and runtime system in three aspects. Firstly, we have modified the code generation of the compiler to provide the required information to the specialisation controller. Secondly, we have implemented hooks in the compiler that allow the specialisation controller to initiate the specialisation of requested functions. And, last but not least, we have implemented the specialisation controller itself as part of the SAC runtime system.

To control the collection and reporting of runtime information, we have added an additional flag to the compiler. The option –rtspec– will enable the required extension to code generation. The produced executable differs from standard executables in mainly three aspects. Firstly, we extend the dynamic dispatch code that is generated for function applications where we cannot statically determine the matching instance. Additionally to dynamically choosing the appropriate instance, the extended dispatch code also files a corresponding specialisation request to the specialisation request queue. As functions are always dispatched statically with respect to the base types of arguments, this information mainly comprises the rank and dimensionality of each argument. Furthermore, we send the index into the global registry that corresponds to the called function. This information is used twofold: It allows us to later identify which entry in the registry to update. More importantly, however, the index can be used as a unique token to identify the function to specialise. We use this token to lookup the information that is required in the communication with the compiler.

Note here that we send that shape information rather blindly. In particular, we do not perform any checks on whether a specialisation is actually necessary at this point. To keep the runtime overhead within the actual program as low as possible, we offload these checks into the specialisation controller.

The specialisation request queue itself is implemented in a fairly standard way using condition variables to signal the availability of requests to the specialisation controllers and mutual exclusion to synchronise concurrent access to both ends of the queue by multiple threads collaboratively sharing the productive computation as well as multiple instances of specialisation controllers to satisfy requests.

Secondly, we reroute all applications of dispatch functions via the dispatch function registry. This way, we are able to dynamically rebind function applications to updated implementations of dispatch functions. All that is required is an update to the function pointer in the registry.

Last but not least, we have also modified static function dispatch in the compiler. If no runtime specialisation is requested, we usually dispatch a function call statically as soon as we can identify a single matching instance. However, such an instance could still be rank-generic or shape-generic. When using runtime specialisation, such a dispatch is not desirable. As we use the dynamic dispatch code to trigger runtime specialisation, an application that has been statically dispatched would never be optimised. Therefore, when runtime specialisation is enabled, we only dispatch a function application statically if we were able to derive full shape knowledge for the arguments, and the matching instance is an exact match for those shapes. In those situations, no further specialisation would be possible.

The second work package in our implementation, the special version of the SAC compiler that creates new specialisations on the fly, turned out to require surprisingly implementation effort as we capitalise on a combination of existing compiler features. To maximise reuse of the existing compiler implementation, we have not created a dedicated interface to the compiler for runtime specialisation. Instead, the dynamic specialisation controller creates the intermediate representation of a standard SAC module. For example, to create a specialised instance of the convolution step function from Figure 3 for a $10 \times 10$ matrix of double values, the intermediate representation corresponding to the module shown in Figure 8 is created.

Essentially, the generated stub module consists of a module (name space) declaration (line 1), an import-statement for the symbol to be specialised (line 3), an export-statement for the new (specialised) symbol to be created (line 5) and, most importantly, a *specialisation directive* (line 7). The import-statement in Line 3 instructs the compiler to load the definition of the function to be specialised into the current name space, that is, the otherwise empty module rtspec_mod_001. We use the name rtspec_mod_001 as an illustrative example. Any name can be used as long as it is unique during the runtime of the application. In particular, no two specialisation requests may use the same module name. Our implementation uses an appropriate strategy to derive such module names.

The second essential component of the generated intermediate representation is the specialisation directive to the compiler in Line 7. This directive is directly generated from the specialisation request data extracted from the queue. As can be seen, large parts of the required data, for example, the return type and the types of non-generic arguments, are identical for all valid specialisation requests. These components of the request can and will be statically preassembled to accelerate the enqueueing of requests.

During this process, the compiler dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SAC module system that was originally developed to support inter-module (compile time) optimisation [13].

```
1 module rtspec_mod_001;
2
3 import Convolution : {convolution_step};
4
5 export {convolution_step};
6
7 specialize double[*] convolution_step( double[10,10] A);
```

Figure 8. Intermediate representation of a stub module used for a specialisation request for the function convolution step from Figure 3.

All that remains to be done to exploit the existing machinery for runtime specialisation is to create the preceding code, at least in form of an abstract syntax tree in memory, start the compilation process and dynamically add the resulting library. This functionality, among other bookkeeping, is implemented in the specialisation controller. In the simplest case, the controller dequeues a specialisation request, creates the corresponding abstract syntax tree to trigger the specialisation, enacts the compiler and collects back the updated library. That library is then dynamically linked to the program, and the global registry is updated.

However, as the augmented program submits specialisation requests blindly, we might end up with many duplicate requests for specialisations that have already been performed. To prevent useless specialisation runs, the controller keeps track of requests it has acted upon and automatically disregards future requests of the same kind. Using this technique, we ensure that each request is only acted upon once.

Our adaptive compilation framework is carefully designed such that the associated runtime overhead in the executable program is minimal. Essentially, it boils down to an indirection in calling the dispatch function and the filing of a specialisation request. All the remaining work is done concurrently to the execution of the program itself by one or more dynamic specialisation controllers. Our essential assumption is that these run on different processors or cores and as such use resources that would otherwise remain unused or whose exploitation for running the program itself would, at most, have a marginally positive effect on overall performance.

## 7. EXPERIMENTAL EVALUATION

Our experimental evaluation is based on the generic programming case study introduced in Section 3. We use an AMD Phenom II X4 965 quad-core system running at 3.4-GHz clock frequency. It is equipped with 4-GB DDR3 memory, and the operating system is Linux with kernel 2.6.38-rc1.

We ran our generic convolution kernel for a total of six different problem sizes. For each problem size, we recorded the execution time of each individual iteration of the convolution using a high resolution real-time clock. We uniformly set the number of iterations to 50 while we use a convergence threshold and initial array values, which guarantee that we effectively run these 50 iterations. Because we read some initialisation data from a file, the SAC compiler is unable to deduce this information statically, and we effectively evaluate the convergence check in every iteration.

To isolate the effect of adaptive compilation more clearly, we refrain from running the application itself with multiple threads for now and only employ a single instance of the specialisation controller throughout the experiments.

Figure 9 shows experimental results for three different matrix sizes: $500 \times 500$, $1000 \times 1000$ and $10,000 \times 5,000$. Each experiment is made with and without runtime specialisation enabled. For all three problem sizes, we can easily identify a recurring pattern in the runtime behaviour. At the beginning, code with and without runtime specialisation enabled takes about the same time per iteration. In principle, code with runtime specialisation enabled should run slightly slower because it contains additional instructions for spawning the specialisation controller thread, identifying potential specialisation cases and enqueuing specialisation requests. In the given example, this overhead of runtime specialisation seemingly is negligible compared with the computational workload of convolution even for the smallest problem size investigated.

In our case study, the first convolution iteration triggers the first specialisation requests for functions convolution step and is convergent. As soon as specialised and, in consequence, far better optimised versions of these functions become available, we can identify a dramatic decrease in single iteration runtimes. As the problem size remains constant throughout each individual program run, no further specialisations occur, and the shorter iteration runtime remains constant until the end of each program run.

The time it takes to dynamically recompile versions of the functions convolution step and is convergent specifically adapted to the individual experimental settings is, of course, independent of these settings in general and the problem size used in particular. Hence, the larger the problem size is, the less convolution iterations we need to wait until adapted code becomes available. For a

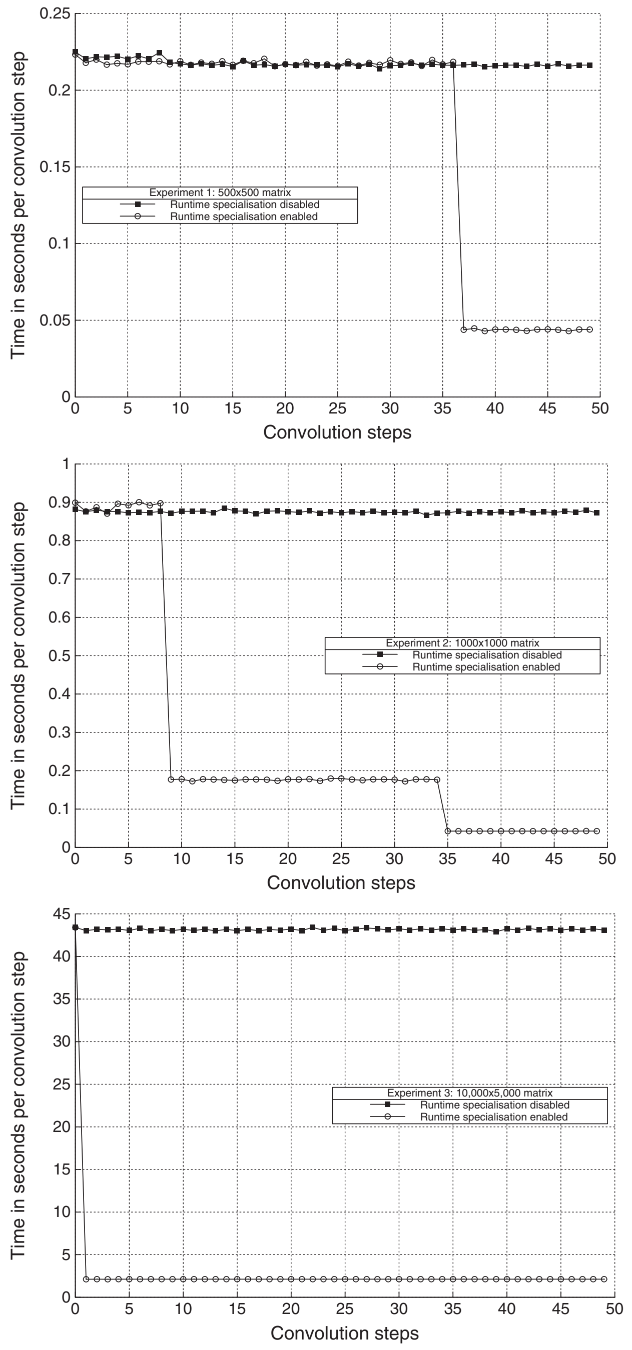


Figure 9. Experimental results obtained by applying the case study code discussed in Section 3 to matrices of different size with runtime specialisation disabled and enabled.

$500 \times 500$ matrix, it takes more than 35 iterations until the adapted version of the convolution step function becomes available. Because we set the maximum number of iterations to 50 in our experiments, we wait in vain for a specialised version of the convergence check. For a $1000 \times 1000$ matrix, we only wait for nine iterations to obtain an optimised convolution step that leads to a more than fivefold performance increase. After another 26 iterations, we also obtain the optimised convergence check, which makes the execution time of a single convolution step drop by another factor of 4. For the largest problem size, a $10\,000 \times 5000$ matrix, a single convolution step is sufficient for the adaptive compilation framework to produce optimised versions of both functions convolution step and is convergent. As a consequence, we observe a 20-fold speedup in execution time. These numbers are in accordance with the relative problem sizes as can be expected from a numerical kernel whose computational complexity is linear in the problem size.

One of the remarkable features of our generic convolution kernel is that it cannot only be applied to matrices of any size but likewise to arrays of different rank, or number of dimensions. In Figure 10, we show further experimental results obtained from applying the convolution kernel to

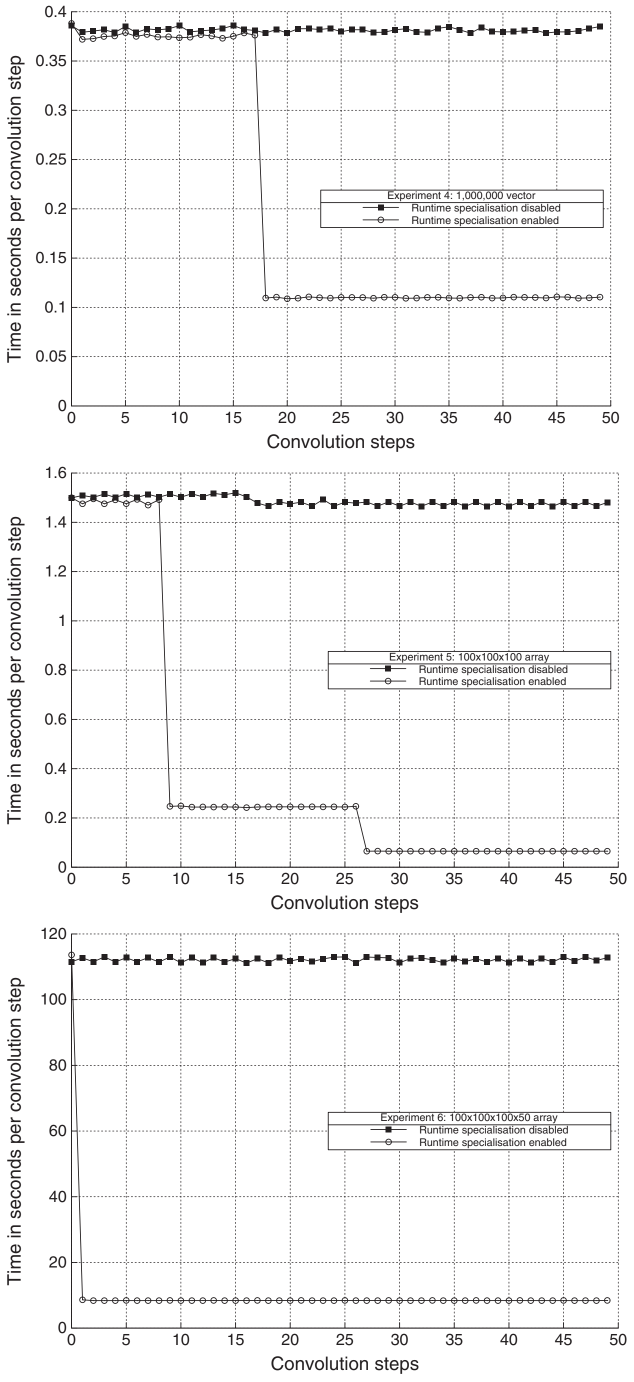


Figure 10. Experimental results obtained by applying the case study code discussed in Section 3 to a vector, a tensor and a four-dimensional array with runtime specialisation disabled and enabled.

a vector of 1 000 000 elements, a tensor of $100 \times 100 \times 100$ elements and a four-dimensional array of $100 \times 100 \times 100 \times 50$ elements. The experiments confirm our observations in Figure 9. Depending on the size of the workload, we are able to materialize significant improvements within few convolution steps.

How representative is a single benchmark? The proposed adaptive optimisation technique capitalises essentially on two aspects: the performance difference between generic and non-generic code and the unavailability of exact shape information of essential arrays at compile time. In our view, this combination rather is the norm in real-world applications than the exception. The first aspect depends on the actual programming style, of course, but as soon as programs follow the advocated style using composition of building blocks and rich abstraction hierarchies, similar effects as in the convolution example are inevitable. The second aspect is just as relevant. As soon as the building of an application is time-wise and person-wise separated from the running of the application, which again rather is the norm in real-world applications than the exception, the opportunities for static specialisation will be extremely limited. Where this separation does not prevail, for example in high-performance computing, often applications internally apply the same functions to arguments of different shape in a way that is difficult to deduce statically even if the initial shapes are indeed known. Examples here are the NAS benchmarks MG and FT [14, 15].

## 8. RELATED WORK

A wealth of related work can be found in the area of runtime partial evaluation, often also referred to as dynamic specialisation. Systems such as Tempo [16, 17], Fabius [18] or DyC [19] are based on user annotations, which indicate to the compiler where dynamic specialisations can be expected. These systems then generate specific runtime specialisers leading to a staged compilation process. This measure keeps the overhead introduced by the compilation at runtime low. In contrast, our approach is based on the idea to specialise programs concurrently and asynchronously. This allows us to apply the full-fledged compiler to an annotated source code.

Further related work concerns binary translation approaches that operate on the code that is being executed. They typically analyse different instruction paths at runtime. When it turns out that a certain path is used frequently, these paths are optimised further.

Dynamo [20] and DynamoRio [21] both identify hot spots in programs. When a hotspot has been identified, execution is paused, and optimised code is generated for it. As interpreting is expensive, Dynamo tries to store as many optimised traces as possible in a trace cache. The next time a trace is executed, Dynamo points it to the optimised code stored in its cache.

Another approach, Adaptive Object Code RE-optimisation (ADORE) [22], uses hardware performance monitoring to identify performance bottlenecks. Similar to the approach presented in this paper, ADORE uses two threads: One thread runs the application as it would have normally, and the second thread runs the optimisation functions. However, the optimisations performed in the ADORE system primarily target insertions of data cache prefetching to improve the cache behaviour in subsequent runs.

The lack of static knowledge about array operations drives other approaches to data-parallel computing towards runtime optimisation as well. In FlumeJava [23], compiled programs do not execute data-parallel programs directly but construct an execution plan at runtime. This execution plan is then further optimised before it is executed. A similar approach is used by Intel's Ct [24] in the context of C++. In contrast to our approach, FlumeJava and Ct are designed for runtime code generation. Neither produces directly executable code. As such, program execution is halted while the just-in-time compiler generates an optimised executable, whereas our approach offloads this to a dedicated core while program execution continues. Furthermore, we use a significantly more heavyweight compilation technology.

## 9. CONCLUSION

We have presented an adaptive compilation framework for generic array programming that virtually achieves the quadrature of the circle: to program code in a generic, reuse-oriented way abstracting from concrete structural properties of the arrays involved and, at the same time, to enjoy the runtime performance characteristics of highly specialised code when it comes to program execution.

As multiple cores are already rather the norm in contemporary processors, and the number of cores is predicted to grow quickly in the near future, adaptive optimisation virtually comes for free. We run all dynamic recompilations/specialisations fully asynchronously with the main computation. Thus, their delaying effect on the main computation is minimal. With the growing number of cores, this observation even holds for computational code that is itself multithreaded. Reserving a small fraction of available cores for adaptive compilation either permanently or temporarily has a minor effect on the computation's progress even for linearly scaling programs.

Extensive experimental data using a highly generic implementation of a highly relevant numerical kernel, multidimensional rank-generic convolution demonstrate the usefulness of our approach. In essence, we effectively use an otherwise unused core of a multicore system to continuously adapt the running code to problem sizes prevailing in a concrete application. We overcome the limitations of static specialisation by recompiling and highly optimising generic code at runtime when structural properties of arrays are fully available.

Because each dynamic invocation of the compiler incurs some time independent of the problem size, adaptive compilation becomes more profitable for long-running applications. In an extreme case, the program itself could run to completion before the first spawned specialisation request is actually satisfied. However, our experimental data suggests that the overhead within the running code is rather small and, hence, the main wasting of resources would be in occupying one core to produce code that is never going to be run. In essence, all numerically interesting/challenging real-world problems can be classified as long-running relative to a compiler invocation.

Likewise, our approach builds on the assumption of temporal locality, that is, the assumption that if some function is applied to some arguments of certain shapes, it will, later during program execution, again be applied to arguments of the same shapes (but most likely different values). If this assumption does not hold for a certain program, adaptive compilation cannot be expected to provide any benefits to runtime performance. In such a case, it should rather be disabled to avoid wasting resources such as the core used for program adaptation. However, the detrimental effect of adaptive compilation on the performance of the program itself is very small.

## 10. FUTURE WORK

It does not take much to identify a wealth of research questions arising from realising the proposed adaptive compilation framework. For example, given a number of available cores, what is a profitable division of cores into one group of cores that collaboratively execute the program and another group of cores that run dynamic specialisation controllers.

Furthermore, it would be more than reasonable to complete more than one specialisation request at a time, but rather take all such requests from the request queue that have been filed since the previous specialisation round. Although the dynamic invocation of the SAC compiler is not on the critical path because of running concurrently with the main program on different computing resources, compiler runtimes are not completely irrelevant either. Therefore, it may be useful to run the SAC compiler with a different option set in these cases and, moreover, to use a specific build of the SAC compiler geared for performance, for example by leaving out all sorts of debug code. The compiler used in the experiments, for instance, still did contain all this; hence, we expect a great potential for speeding up dynamic invocations.

For now, we only collect specialisations during one program run. However, the same specialisations are likely to be helpful across multiple invocations of the same program or even across programs sharing some set of basic libraries, for example the SAC standard library. Hence, it would be desirable to persistently store specialised binary code along with the generic binary code of the original module implementation. Thus, we would, over time, create a growing base of prespecialised instances of certain functions ready for use in subsequent program runs. Such a persistent module storage creates a further number of interesting and challenging research question. For example, we cannot safely let the persistent modules grow only at some point, we also need to discard existing function instances, which require a sort of replacement policy.

Last but not least, dynamic specialisation only makes sense for functions that actually benefit from the availability of more detailed structural information on argument arrays. This definitely holds for computationally intensive functions, but not so much, for example, for I/O-related functions. Identifying suitable functions alone is an interesting future research question. This is somewhat related to hotspot detection in standard jit-compilation, but still has its very own characteristics. For example, in this case, we may actually get very far with static code analysis techniques, whereas hotspot detection rather requires some form of runtime profiling.

## ACKNOWLEDGEMENT

This work is partially supported by the European Union through the STREP project Advance (Asynchronous and Automatic Virtualisation through Performance Analysis to Support Concurrency Engineering, project number FP7 248828).

## REFERENCES

1. Fursin G, Cohen A, *et al*. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*. Springer Verlag: Berlin, Heidelberg, Germany, 2005; 29–46.
2. Grelck C, Scholz SB. Merging compositions of array skeletons in SAC. In *Parallel Computing: Current and Future Issues of High-End Computing, International Conference ParCo 2005, Malaga, Spain, NIC Series*, Vol. 33, Joubert G, Nagel W, Peters F, Plata O, Tirado P, Zapata E (eds). John von Neumann Institute for Computing: Jülich, Germany, 2006; 859–866. [ISBN 3-00-017352-8].
3. Herhut S, Scholz SB, *et al*. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In *Implementation and Application of Functional Languages, 19th International Symposium, IFL'07, Freiburg, Germany, Revised Selected Papers, Lecture Notes in Computer Science*, Vol. 5083, Chitil O, Horváth Z, Zsók V (eds). Springer-Verlag: Berlin, Heidelberg, 2008; 254–273.
4. Bernecky R, Herhut S, *et al*. Symbiotic expressions. In *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Revised Selected Papers, Lecture Notes in Computer Science*, Vol. 6401, Morazán MT, Scholz SB (eds). Springer-Verlag: Berlin, Heidelberg, 2010; 107–124.
5. Grelck C, Scholz SB. SAC: a functional array language for efficient multithreaded execution. *International Journal of Parallel Programming* 2006; **34**(4):383–427.
6. Kreye D. A compilation scheme for a hierarchy of array types. In *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers, Lecture Notes in Computer Science*, Vol. 2312, Arts T, Mohnen M (eds). Springer-Verlag: Berlin, Heidelberg, Germany, 2002; 18–35.
7. Grelck C, Scholz SB. SAC — from high-level programming with arrays to efficient parallel execution. *Parallel Processing Letters* 2003; **13**(3):401–412.
8. Grelck C, Scholz SB. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing* 2006; **32**(7–8):507–522.
9. Grelck C. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 2005; **15**(3):353–401.
10. Marcussen-Wulff N, Scholz SB. On interfacing SAC modules with C programs. In *12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany, Aachener Informatik-Berichte*, Vol. AIB-00-7, Mohnen M, Koopman P (eds). Technical University of Aachen: Aachen, Germany, 2000; 381–386.
11. Grelck C, Scholz S, *et al*. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming* 2010; **38**(1):38–67. DOI: 10.1007/s10766-009-0121-x.
12. Grelck C, Trojahner K. Implicit memory management for SaC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04, Lübeck, Germany*, Grelck C, Huch F (eds). University of Kiel, Institute of Computer Science and Applied Mathematics: Kiel, Germany, 2004; 335–348. Technical Report 0408.
13. Herhut S, Scholz SB. Towards fully controlled overloading across module boundaries. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Grelck C, Huch F (eds). University of Kiel, Institute of Computer Science and Applied Mathematics: Kiel, Germany, 2004; 395–408. Technical Report 0408.
14. Grelck C. Implementing the NAS Benchmark MG in SAC. In *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*, Prasanna VK, Westrom G (eds). IEEE Computer Society Press: Los Alamitos, California, USA, 2002.
15. Grelck C, Scholz SB. Towards an efficient functional implementation of the NAS Benchmark FT. In *Parallel Computing Technologies, 7th International Conference, PaCT'03, Nizhni Novgorod, Russia, Lecture Notes*

*in Computer Science*, Vol. 2763, Malyshkin V (ed.). Springer-Verlag: Berlin, Heidelberg, Germany, 2003; 230–235.

16. Consel C. A general approach for run-time specialization and its application to C. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, USA*. ACM Press: New York City, New York, USA, 1996; 145–156.
17. Noel F, Hornof L, Consel C, Lawall JL. Automatic, template-based run-time specialization: implementation and experimental study. In *International Conference on Computer Languages*. IEEE Computer Society Press: Los Alamitos, California, USA, 1998; 132–142.
18. Leone M, Lee P. Dynamic specialization in the Fabius system. *ACM Computing Surveys* 1998; **30**(3es).
19. Grant B, Philipose M, *et al*. An evaluation of staged run-time optimizations in DyC. *ACM SIGPLAN Notices* 1999; **34**(5):293–304.
20. Bala V, Duesterwald E, *et al*. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.* May 2000; **35**(5):1–12. http://doi.acm.org/10.1145/358438.349303.
21. Bruening D, Garnett T, *et al*. An infrastructure for adaptive dynamic optimization. *International Symposium on Code Generation and Optimization*, March 2003.
22. Lu J, Chen H, *et al*. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism* April 2004; **6**:1–24.
23. Chambers C, Raniwala A, *et al*. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10, ACM: New York, NY, USA, 2010; 363–375.
24. Ghuloum A, Smith T, *et al*. Future-proof data parallel algorithms and software on Intel multi-core architecture. *Intel Technology Journal* November 2007; **11**(4):333-347. DOI: 10.1535/itj.1104.07.