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

Clemens Grelck Heinz Wiesinger
University of Amsterdam
Institute of Informatics
C.Grelck@uva.nl H.M.Wiesinger@student.uva.nl

Abstract

Data-parallel processing of multi-dimensional arrays is characterised by a fundamental trade-off between software engineering principles on the one hand and runtime performance concerns on the other hand. Whereas the former demand code to be written in a generic style abstracting from structural properties of arrays as much as possible, the latter require an optimising compiler to have as much information on the very same structural properties available at compile time. Asynchronous adaptive specialisation of generic code to specific data to be processed at application runtime has proven to be an effective way to reconcile these contrarian demands.

In this paper we revisit asynchronous adaptive specialisation, provide a comprehensive analysis of its strengths and weaknesses and propose improvements for its design and implementation. These improvements are primarily concerned with making specialisations available to running applications as quickly as possible. One important measure is making specialisations persistent across multiple invocations of the same or even different applications. Another one is to run multiple specialisation attempts concurrently taking further advantage of today's multi-core chip architectures.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Array processing, Single Assignment C, runtime optimisation, dynamic recompilation

1. Introduction

SAC (Single Assignment C) is a purely functional, data-parallel array programming language [3, 5]. As such, SAC puts the emphasis on homogeneous, multi-dimensional arrays as the most relevant data aggregation principle. SAC advocates shape- and rank-generic programming on multi-dimensional arrays, i.e. SAC supports functions that abstract from the concrete shapes and even from the concrete ranks (number of dimensions) of argument and result arrays. Depending on the amount of compile time *structural* information we distinguish between different runtime representations of arrays.

From a software engineering point of view it is (almost) always desirable to specify functions on the most general input type(s) to maximise code reuse. For example, a simple structural operation like rotation should be written in a rank-generic way, a naturally rank-specific function like an image filter in a shape-generic way. Very infrequently it is desirable to write code in a non-generic way. Consequently, the extensive SAC standard library is full of generic, mostly rank-generic functions.

However, genericity comes at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again for rank-generic code [15]. The reasons are manifold and often operation-specific, but three categories can be identified nonetheless: Firstly, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Secondly, many of the SAC compiler's advanced optimisations [6, 7] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Thirdly, in automatically parallelised code [1, 4, 10] many organisational decisions must be postponed until runtime and the ineffectiveness of optimisations inflicts frequent synchronisation barriers and superfluous communication.

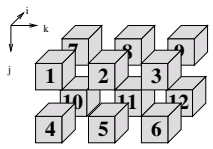
In order to reconcile the desires for generic code and high runtime performance, the SAC compiler aggressively specialises rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analysis for rank and shape specialisation, this approach is fruitless if the necessary information is not available at compile time as a matter of principle. For example, the corresponding data may be read from a file, or the SAC code may be called from external (non-SAC) code, to mention only two potential scenarios.

Such scenarios and the ubiquity of multi-core processor architectures form the motivation for our asynchronous adaptive specialisation framework [8, 9]. The idea is to postpone specialisation, if necessary, until runtime time, when full structural information is always available. Asynchronous with the execution of a generic function, potentially in a data-parallel fashion on multiple cores, a *specialisation controller* generates an appropriately specialised binary variant of the same function and dynamically links the additional code into the running application program. Eligible functions are indirectly dispatched such that if the same binary function is called again with arguments of the same shapes as previously, the corresponding new and fast non-generic clone is run instead of the old and slow generic one.

In contrast to standard just-in-time compilation approaches for (byte code) interpreted languages we take advantage of today's ubiquity of multi-core architectures and the continuously growing number of available cores in average computing environments. With asynchronous adaptive specialisation the re-compilation of specialised intermediate code happens in parallel with the running application. The rationale here is that, with a large number of cores, having one core less available for data-parallel program execution typically has a negligible effect on runtime performance, if any.

IFL'13, August 28–30, 2013, Nijmegen, Netherlands.

ACM ???.
http://dx.doi.org/10.1145/???



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

The effectiveness of our approach, in general, depends on making specialised and thus considerably more efficient binary variants available to a running application as quickly as possible. The contribution of this paper is to investigate optimisations and extensions of our framework proposed in [8, 9] to this effect. These extensions fall into two categories. Our first approach is to parallelise the specialisation controller in order to produce multiple specialisations concurrently. Instead of a fixed classification of the host architecture's cores as either compute cores or specialisation cores, we propose a demand-driven dynamic adjustment of hardware resources. Our second approach is to make specialisations persistent across multiple runs of the same application or even across multiple unrelated applications that make use of an overlapping set of libraries.

The remainder of the paper is organised as follows. In Section 2 we explain SAC in general and the calculus of multi-dimensional arrays in more detail; in Section 3 we do the same for the runtime specialisation framework. A comprehensive analysis of the strengths and weaknesses follows in Section 4. Based on this we propose parallel runtime specialisation in Section 5 and persistent runtime specialisation in Section 6. Finally, we sketch out some related work in Section 7 and draw conclusions in Section 8.

2. SAC and the Calculus of Multi-Dimensional Arrays

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 choice is primarily 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 can be found in [3, 5].

Following the example of interpreted array languages, such as APL[2, 11], J[12] or NIAL[13, 14], an array value in SAC is characterised by a triple $(r, \vec{s}, \vec{d})$. The *rank* $r \in \mathbb{N}$ defines the number of dimensions (or axes) of the array. The *shape vector* $\vec{s} \in \mathbb{N}^r$ yields the number of elements along each of the r dimensions. The *data vector* $\vec{d} \in T^{\prod \vec{s}}$ contains the array elements (in row-major unrolling), the so-called *ravel*. Here T denotes the element type of the array. Some relevant invariants ensure the consistency of array values. The rank equals the length of the shape vector while the product of the elements of the shape vector equals the length of the data vector.

Fig. 1 illustrates the calculus of multi-dimensional arrays that is the foundation of array programming in SAC. The array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Consequently, every value in SAC has rank, shape vector and data vector. Both rank and shape vector can be queried by built-in functions of the same name. The data vector can only be accessed element-wise through a selection facility adopting the square bracket notation familiar from C-like languages. Given the ability to define rank-generic functions, whose argument array's ranks may not be known at compile time, indexing in SAC is done using vectors (of potentially statically unknown length), not (syntactically) fixed sequences of scalars as in most other languages. Characteristic for the calculus of multi-dimensional arrays is a complete separation between data assembled in an array and the structural properties (rank and shape) of the array.

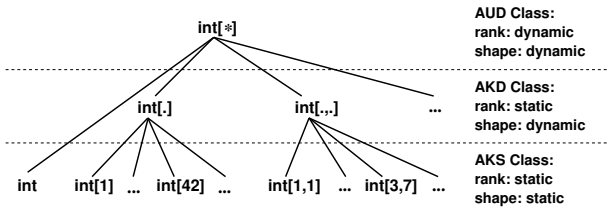


Figure 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

The type system of SAC is monomorphic in the element type of an array, but polymorphic in the structure of arrays. As illustrated in Fig. 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, e.g. `int[3,7]` or just `int`. On an intermediate level of genericity we find arrays of fixed rank, e.g. `int[.,.]`. And on the top of the hierarchy we find arrays of any rank, e.g. `int[*]`. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.

The array type system leads to three different runtime representations of arrays depending on the amount of compile time structural information, as illustrated in Fig. 2. For *AKS arrays* both rank and shape are compile time constants and, thus, only the data vector is carried around at runtime. For *AKD arrays* the rank is a compile time constant, but the shape vector is fully dynamic and, hence, must be maintained alongside the data vector. Last not least, for *AUD arrays* both shape vector and rank are dynamic.

3. Asynchronous Adaptive Specialisation

In order to reconcile software engineering principles for generality with performance demands we have developed the asynchronous adaptive specialisation framework illustrated in Fig. 3. The idea is to postpone specialisation if necessary until runtime time, when all structural information is eventually available no matter what. A generic SAC function compiled for runtime specialisation leads to two functions in binary code: the original generic and presumably slow function definition and a small proxy function that is actually called by other code instead of the generic binary code.

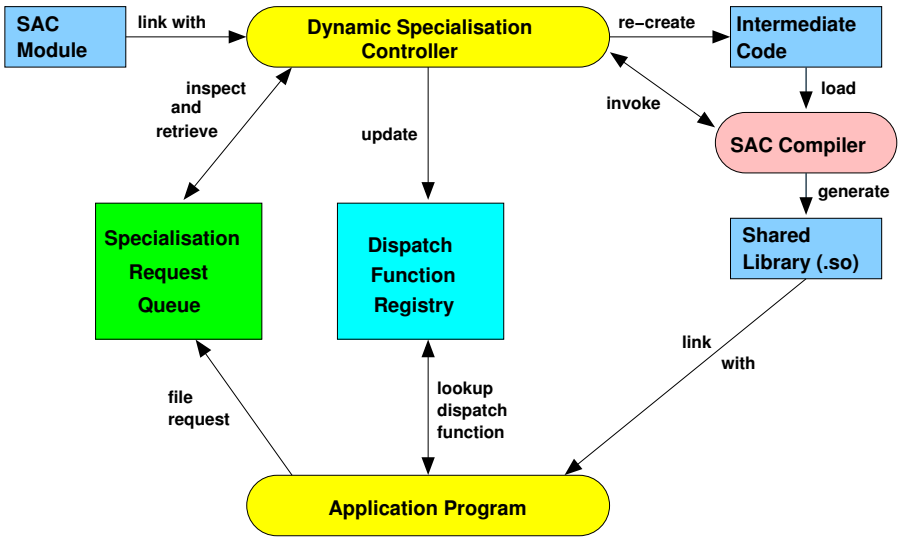


Figure 3. Software architecture of asynchronous adaptive specialisation framework

When executed, the proxy function files a specialisation request consisting of the name of the function and the concrete shapes of the argument arrays before calling the generic implementation. Of course, proxy functions also check whether the desired specialisation has been built before, or whether an identical request is currently pending. In the former case, the proxy function dispatches to the previously specialised code, in the latter case to the generic code, but without filing another request.

Concurrent with the running application, a specialisation controller (thread) takes care of specialisation requests. It runs the fully-fledged SAC compiler with some hidden command line arguments that describe the function to be specialised and the specialisation parameters in a way sufficient for the SAC compiler to re-instantiate the function's partially compiled intermediate code from the corresponding module, compile it with high optimisation level and generate a new dynamic library containing the specialised code and a new proxy function. Eventually, the specialisation controller links the application with that library and replaces the proxy function in the running application.

4. Analysis

The effectiveness of asynchronous adaptive specialisation depends on how often the dynamically specialised variant of some function is actually run instead of the original generic version. This depends on two connected but distinguishable properties. Firstly, the application itself must apply an eligible function repeatedly to arguments with the same shape. Secondly, the specialised variant must become available sufficiently quickly to have a relevant impact on application performance. In other words, the application must run considerably longer than the compiler needs to generate binary code for specialised functions.

The first condition relates to a property of the application. Many applications in array processing do expose the desired property, but obviously not all. We can only deal with unsuitable applications by dynamically analysing an application's properties and by stopping the creation of further specialised functions at some point.

The second condition sets the execution time of application code in relation to the execution time of the compiler. In array programming, however, the former often depends on the size of the arrays being processed, whereas the latter depends on the size and structure of the intermediate code. Obviously, execution time and compile time of any code are unrelated with each other and, thus, many scenarios are possible.

```
1  module ConvolutionAuxiliaries;
2
3  use Array: all;
4
5  export {convolution_step, is_convergent};
6
7  double[*]
8  convolution_step (double[*] A)
9  {
10   R = A;
11
12   for (i=0; i<dim(A); i++) {
13     R = R + rotate( i,  1, A)
14           + rotate( i, -1, A);
15   }
16
17   return R / tod( 2 * dim(A) + 1);
18 }
19
20 bool
21 is_convergent (double[*] new,
22                double[*] old,
23                double epsilon)
24 {
25   return all( abs( new - old) < epsilon);
26 }
```

Figure 4. Case study: generic convolution step and convergence check

We demonstrate potential runtime behaviour of applications by means of a small case study: rank-generic convolution with convergence check. Fig. 4 shows the definition of a SAC module `ConvolutionAuxiliaries` that defines and exports two rank-generic functions: `convolution_step` and `is_convergent`. The former defines a single convolution step that computes each element of a multi-dimensional grid as the arithmetic mean of its direct neighbours along each axis. The latter implements a predicate whether or not all elements of a two given arrays differ by less than

a given threshold. Due to using the `rotate` function imported from the comprehensive SAC array library this concolution step implements cyclic boundary conditions.

A more detailed description of the compositional style of array programming advocated by SAC along with a more thorough explanation of a variant of the code shown here can be found in [3]. Since this is not a paper about programming in SAC or the language design of SAC, we refrain from repeating this information here.

```
1  module Convolution;
2
3  use Array: all;
4
5  import ConvolutionAuxiliaries: all;
6
7  export {convolution};
8
9  double[*]
10 convolution (double[*] A, double epsilon)
11 {
12   A_new = A;
13
14   do {
15     A_old = A_new;
16     A_new = convolution_step( A_old);
17   }
18   while (!is_convergent( A_new, A_old,
19                          epsilon));
20   return A_new;
21 }
```

Figure 5. Case study: generic convolution kernel with convergence check

Fig. 5 shows a second module named `Convolution`. This module defines and exports a single function named `convolution`, which computes a series of convolution steps until sufficient convergence is reached. More precisely, in every iteration (tail recursion) the function `convolution` applies both imported functions `convolution_step` and `is_convergent`.

We compile the module `ConvolutionAuxiliaries` (Fig. 4) with and without runtime specialisation enabled and import either version into the module `Convolution`. We conducted a series of experiments with different array ranks and shapes on an AMD Phenom II X4 965 quad-core system. The machine runs at 3.4GHz clock frequency and is equipped with 4GB DDR3 memory. The operating system is Linux with kernel 2.6.38-rc1.

A representative plot of the runtimes achieved is shown in Fig. 6; it reports on a convolution experiment with a 3-dimensional array of $100 \times 100 \times 100$ double precision floating point numbers. The figure shows individual iterations on the x-axis and measured execution time for each iteration on the y-axis. The two lines show measurements with runtime specialisation disabled and enabled, respectively.

The first insight is that for the given example runtime specialisation does not inflict any measurable overhead in the startup phase while the specialisation controller is still working on the first specialisation.

After 8 iterations running completely generic binary code a shape-specialised version of the `convolution_step` function becomes available. Switching from a generic to a non-generic implementation of the convolution step reduces the execution time per iteration from about 1.5 seconds to roughly 0.25 seconds. This example demonstrates the tremendous effect that runtime specialisation can have on generic array code. The 3-dimensional case requires a total of six rotations of the argument array. Rotation is not a built-in function in SAC, but itself is implemented using two consecutive basic array operations (with-loops). Rank-generic binary code cannot further be optimised and leads to a total of 19 intermediate arrays to compute the final result. For the specialised intermediate code the compiler manages to unroll the for-loop three times and to fold the resulting array operations such that the entire convolution step is computed in one step without any intermediate temporary structures materialising in memory.

As soon as the specialisation of the convolution step is completed, the specialisation controller starts working on the already pending specialisation request for the convergence check. As illustrated in Fig. 6, the specialised binary code for the convergence check becomes available after 26 iterations and reduces the execution time of a single iteration further from 0.25 seconds to 0.065 seconds. The main reason for this considerable performance improvement again is the effectiveness of optimisations that fuse consecutive array operations and, thus, avoid the creation of intermediate arrays. With all binary code specialised for the relevant array shape $100 \times 100 \times 100$ no further improvements are to be expected for the remainder of the application runtime.

As pointed out earlier, the time it takes to make a specialised binary variant of either the convolution step or the convergence check available to the running application is constant (for a given intermediate code, compiler version and compiler options). In contrast, the time it takes to complete one iteration of the application depends on the rank and shape of the argument array. Running the very same application on a considerably larger argument array may lead to a situation in which both specialisations become available long before the application moves on to the second iteration. The other extreme is likewise possible. For small arrays the entire application may have terminated before even the first specialisation becomes available. In this case the specialisation controller discards the specialisation attempt and terminates alongside the application itself. As a consequence, the entire application runs generic binary code just as without runtime specialisation.

These scenarios illustrate that runtime specialisation is most effective for long-running applications. Furthermore, they demonstrate that any measure that contributes to making specialised binary variants quicker available to a running application is beneficial in practice and improves the applicability of the entire approach.

5. Parallel Asynchronous Adaptive Specialisation

A number of aspects affect the time that it takes from filing a specialisation request by the running application to the specialised binary effectively becoming available for dispatch. The most relevant aspect in one way or another is the execution speed of the compiler. For good reasons the design of the SAC compiler is diametrically opposed to that of typical just-in-time compilers for bytecode interpreted languages. Whereas the latter are optimised for short compilation times, the SAC compiler has from the very beginning been optimised for speed of compiled code, not speed of the compiler.

Many different large-scale code transformations/optimisations contribute to this design at the expense of considerable compilation times even for relatively short source codes. Of course, the most time-consuming optimisations could be switched off for the runtime specialisation use case, but this would be counter-productive. It is exactly this optimisation capacity that is essential for achieving the substantial performance gains through specialisation, as demonstrated in the previous section. Thus, speeding up asynchronous adaptive specialisation in general is limited and would require a long-term engineering investment.

What can be achieved with much less effort and, hence, is at the core of our proposed improvements, is the parallelisation of the dynamic specialisation process itself. With compute cores

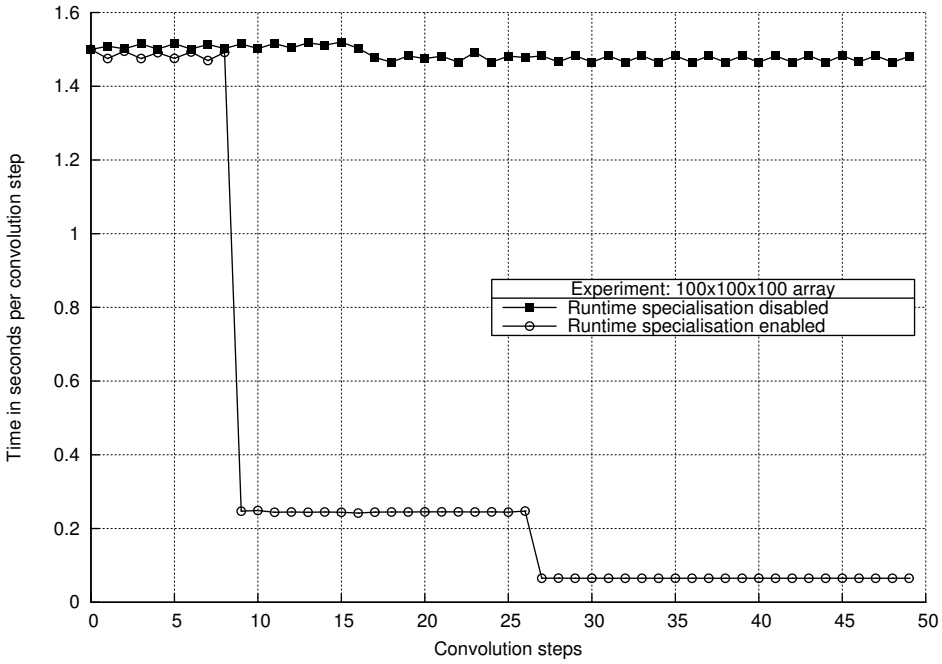


Figure 6. Case study: running the generic convolution kernel defined in Fig. 4 and Fig. 5 on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialisation

available in abundance in the near future, if not already today, the same argument that we used to motivate setting aside one core for specialisation instead of data-parallel execution of the application holds for more than one core.

Looking at Fig. 6 shows that the relative performance improvements realised by adaptive asynchronous specialisation by far outweigh potential improvements through data-parallel execution even when assuming linear speedups and even on the fairly small quad-core system that we used for our experiments. On system with tens of cores parallel specialisation through multiple concurrent specialisation controllers should be beneficial even if the relative performance improvements are less impressive.

For our running example of generic convolution it is fairly clear that two specialisation controllers would be optimal. One would then specialise the convolution step while the other could concurrently specialise the convergence check. According to Fig. 6 the former takes about 12 seconds (8 iterations of 1.5 seconds each) while the latter takes about 5 seconds (18 iterations of 0.25 seconds plus some share of last slow iteration). Looking at the different complexities of the definitions of the convolution step and the convergence check, as shown in Fig. 4, these numbers appear plausible.

In other words, it proves to be rather unfortunate that we first specialise the convolution step and only after completing this task turn towards the convergence check. If we would specialise the convergence check first, partially specialised code would already be available after 3–4 iterations. Unfortunately, the specialisation order is beyond our control because the generic implementation of the convolution step is simply run before the that of the convergence check in the application code.

In any case, with two concurrent specialisation controllers we can expect that the specialised convergence check becomes available after only 3–4 iterations while the specialised convolution step still becomes available in exactly the same time as with a single specialisation controller. Of course, due to the specialised convergence check we would already have computed more iterations at this point in time than before.

While parallelising asynchronous adaptive specialisation appears to be beneficial no matter what if only sufficiently many compute cores are available, the question arises how many cores would be best to use for specialisation and how many for data-parallel execution of the application program itself. For the running example this question seems to be straightforward to answer: two. However, even for this admittedly simple demo application this is not the optimal number. Once both specialisations have been created, the two specialisation controllers would wait in vain for any other requests to come and thus would waste two compute cores until the termination of the application. It seems plausible that these two cores should rather help running the application, in particular on a small quad-core system as we used for experimentation.

Starting out with some default ratio, the expectation is that an application initially requires more specialisations while in many cases a fixed point is reached after some time or at least the need for specialisations reduces as the application continues to run. Thus, we propose to adapt the number of specialisation controllers to the actual demand and leave as many cores as possible to the (implicitly) parallelised application.

6. Persistent Asynchronous Adaptive Specialisation

Another major area of refinement lies in making asynchronous adaptive specialisations persistent. So far specialisations are accumulated during the execution of an application, but are automatically removed upon termination. Consequently, any follow-up run of the same application program starts again from scratch. Of course, the next run may use arrays of different shape, but in many scenarios it is quite likely that a similar set of shapes will prevail as in previous runs.

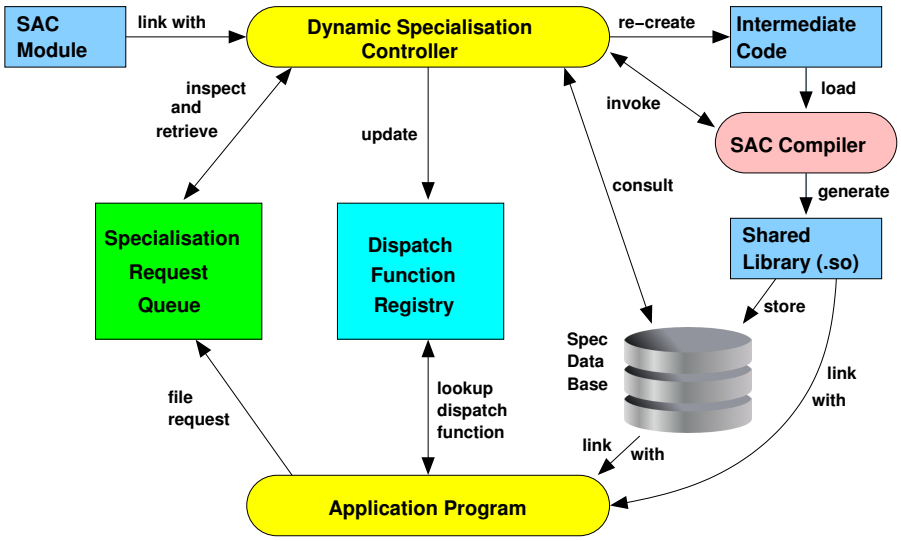


Figure 7. Software architecture of asynchronous adaptive specialisation framework with persistent storage

Therefore, we propose to store specialised binary functions in persistent collections alongside the original generic binary modules. Fig. 7 shows a sketch of the extended framework architecture. The most notable difference to Fig. 3 is the specialised functions data base shown in the lower right corner. Whenever a new specialised binary version of some generic function is created, it is not only linked into the running application that requested this particular specialisation, but it is additionally stored in the data base.

The other area that needs refinement is the specialisation controller. Instead of checking only for potentially existing specialisations created previously in the same application run or currently pending, the specialisation controller additionally consults the external persistent data base to figure out whether or not the required specialisation already exists. Depending on the outcome of this query the application either dispatches to the specialised implementation or files a specialisation request to be taken care of by a specialisation controller.

The main advantage of persistent storage is that the overhead of actually compiling specialisations at application runtime can often be avoided. Our assumption is that for many applications the proposed approach results in a sort of training phase in practice after which most required specialisations have become available. Only in case the user runs an application on a previously unseen array shape does the dynmaic specialisation machinery become active again.

A potential scenario could be image filters. They can be applied to any image pixel format. In practice, however, users only deal with a fairly small number of different image formats. Still, the concrete formats are unknown at compile time. Purchsing a new camera may introduce further image formats to be used. This scenario would result in a short training phase until all image filters have been specialised for the additional image formats of the new camera.

Persistence, however, also creates a new range of research questions. For instance, specialisation repositories cannot grow ad infinitum. We propose to employ statistical methods like *least recently used* or *least often used* to decide when which specialisations may be displaced by new ones. In other words, persistent storage is managed like a cache memory for specialisations.

7. Related Work

Our approach differs from just-in-time compilation of (Java-like) byte code in several aspects. In the latter hot spots of byte code are adapted to the platform they run on by generating native code at runtime while the execution platform was deliberately left open at compile time. This form of adaptation (conceptually) happens in a single step. In contrast, our approach adapts code not to its execution environment but to the data it operates on. This is an incremental process that may or may not reach a fixed point. The number of different array shapes that a generic operation could be confronted with is in principle unbounded, but in practice the number of different array shapes occurring in a concrete application is often fairly limited. Our approach is not specific to SAC, but can be carried over to any context of data-parallel array processing.

8. Conclusions and Future Work

Asynchronous adaptive specialisation is a viable approach to reconcile the desire for generic program specifications in (functional) array programming with the need to achieve competitive runtime performance under limited compile time information about the structural properties (rank and shape) of the arrays involved. This scenario of unavailability of shapely information at compile time is extremely relevant. Beyond potential obfuscation of shape relationships in user code data structures may be read from files or functional array code could be called from less information-rich environments in multi-language applications. Furthermore, the scenario is bound to become practice whenever application programmer and application user are not identical, which rather is the norm than the exception in (professional) software engineering.

In this paper we have proposed several improvements and extensions to asynchronous adaptive specialisation that generally broaden its applicability by making specialised binary code available sooner. The parallelisation of the specialisation process itself with a variable distribution of cores between specialisation and data-parallel application execution allows us to satisfy specialisation requests as quickly as possible. Persistent asynchronous adaptive specialisation aims at sharing runtime overhead across several runs of the same application or even across multiple independent applications sharing some library code and thus to effectively eliminate the observable overhead in many situations.

We are currently working on implementing both parallel and persistent asynchronous adaptive specialisation. Our future work, hence, is dominated by completing this implementation and conducting extensive experiments to evaluate the benefits of the proposed extensions.

References

[1] M. Diogo and C. Grelck. Heterogenous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*. Springer, 2013. to appear.

[2] A. Falkoff and K. Iverson. The Design of APL. *IBM Journal of Research and Development*, 17(4):324–334, 1973.

[3] C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.

[4] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[5] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[6] C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[7] C. Grelck and S.-B. Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[8] C. Grelck, T. van Deurzen, S. Herhut, and S.-B. Scholz. An Adaptive Compilation Framework for Generic Data-Parallel Array Programming. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*. Vienna University of Technology, Vienna, Austria, 2010.

[9] C. Grelck, T. van Deurzen, S. Herhut, and S.-B. Scholz. Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. *Concurrency and Computation: Practice and Experience*, 24(5):499–516, 2012.

[10] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the gpu programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.

[11] International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.

[12] K. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.

[13] M. Jenkins. Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. *Software Practice and Experience*, 19(2):111–126, 1989.

[14] M. Jenkins and J. Glasgow. A Logical Basis for Nested Array Data Structures. *Computer Languages Journal*, 14(1):35–51, 1989.

[15] D. Kreye. A Compilation Scheme for a Hierarchy of Array Types. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2002.