Selected Issues in Persistent Asynchronous Adaptive Specialization for Generic Functional Array Programming

Grelck, C.; Wiesinger, H.

Publication date
2014
Document Version
Submitted manuscript
Published in
IFL 2014: Boston, MA

Selected Issues in Persistent Asynchronous Adaptive Specialization for Generic Array Programming

Clemens Grelck Heinrich Wiesinger

University of Amsterdam
Informatics Institute
Amsterdam, Netherlands
C.Grelck@uva.nl H.M.Wiesinger@student.uva.nl

Abstract

Asynchronous adaptive specialization of rank- and shape-generic code for processing immutable (purely functional) multi-dimensional arrays has proven to be an effective technique to reconcile the desire for abstract specifications with the need to achieve reasonably high performance in sequential as well as in automatically parallelized execution. Since concrete rank and shape information is often not available as a matter of fact until application runtime, we likewise postpone the specialization and in turn aggressive optimization of generic functions until application runtime. As a consequence, we use parallel computing facilities to asynchronously and continuously adapt a running application to the structural properties of the data it operates on.

A key parameter for the efficiency of asynchronous adaptive specialization is the time it takes from requesting a certain specialization until this specialization effectively becomes available within the running application. We recently proposed a persistence layer to effectively reduce the average waiting time for specialized code to virtually nothing In this paper we revisit the proposed approach in greater detail. We identify a number of critical issues that partly have not been foreseen before. Such issues stem among others from the interplay between function specialization and function overloading as well as the concrete organization of the specialization repository in a persistent file system. We describe the solutions we have adopted for the various issues identified.

Categories and Subject Descriptors Software and its engineering [*Software notations and tools*]: Dynamic compilers

Keywords Array processing, Single Assignment C, runtime optimization, dynamic compilation, rank and shape specialization

1. Introduction

SAC (Single Assignment C) is a purely functional, data-parallel array language [4, 6, 7] with a C-like syntax (hence the name). SAC features immutable, homogeneous, multi-dimensional arrays and supports both shape- and rank-generic programming: SAC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions).

In software engineering practice, it is generally desirable to abstract as much as possible from concrete shapes and ranks. This is particularly true for the compositional array programming style advocated by SAC, where in the tradition of APL entire applications are constructed abstraction layer by abstraction layer from basic building blocks, which are by definition rank- and shape-generic as well as application-agnostic.

However, generic array programming 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 [18]. There are various reasons for this observation and often their relative importance is operation-specific, but nonetheless we can identify three categories of overhead caused by generic code: First, 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. Second, many of the SAC compiler's advanced optimizations [8, 9] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Third, in automatically parallelized code [1, 3, 5, 13] many organizational decisions must be postponed until runtime, and the ineffectiveness of optimizations inflicts frequent synchronization barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC compiler aggressively specializes 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 specialization, 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 specialization framework [11, 12]. The idea is to postpone specialization, if necessary, until runtime, when complete structural information on array arguments (rank and shape) is trivially available. Asynchronous with the execution of a generic function, potentially in a data-parallel fashion on multiple cores, a *specialization controller* generates an appropriately specialized binary variant of the same

IFL'14, October 1–3, 2014, Boston, MA, USA.

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

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.

The effectiveness of our approach, in general, depends on making specialized, and thus considerably more efficient, binary variants available to a running application as quickly as possible. This would normally call for fast and light-weight just-in-time compiler, but firstly the SAC compiler is everything but light-weight and rewriting it in a more light-weight style would be a gigantic engineering effort. Secondly, making the compiler faster would inevitably come at the expense of reducing its aggressive optimization capabilities, which obviously is adverse to our overarching goal: highest possible application performance.

In [10] we proposed a total of four different refinements of the original asynchronous adaptive specialization framework:

- bulk asynchronous adaptive specialization,
- prioritized asynchronous adaptive specialization,
- parallel asynchronous adaptive specialization and
- persistent asynchronous adaptive specialization

All four mutually orthogonal techniques aim at reducing the average effective time that it takes for a specialization to become available to the running application once it has been identified as needed.

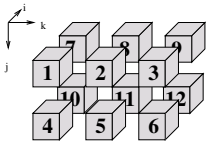
In this paper we focus on the persistence refinement. In the original asynchronous adaptive specialization framework specializations are accumulated during one execution of an application and are automatically removed upon the application's termination. Consequently, any follow-up run of the same application program starts again from scratch as far as specializations are concerned. 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. The same holds across different application programs, in particular as any SAC application is heavily based on the foundation of SAC's comprehensive standard library of rank-generic array operations.

With the proposed persistent storage of specialized functions the overhead of actually compiling specializations at application runtime can often be avoided entirely. For many applications the persistent storage of specializations would in practice result in a sort of training phase, after which most required specializations have been compiled. Only in case the user runs an application on a not previously encountered array shape, does the dynamic specialization 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 of the image processing application. Purchasing 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 specialized for the additional image formats of the new camera.

However, persistence requires more radical changes to the dynamic specialization framework than thought at first glance. This paper is about these issues and how to solve them.

The remainder of the paper is organized as follows. In Section 2 we explain SAC in general and the calculus of multi-dimensional arrays in particular. In Section 3 we elaborate on the existing runtime specialization framework in more detail. Through Sections 4–7 we sketch out a number of issues that arise from the desire to make specializations persistent and explain how to solve them. Finally, we draw conclusions in Section 8.



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

2. SAC and its 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 familiarization 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 [4, 7].

Following the example of interpreted array languages, such as APL[2, 14], J[15] and NIAL[16, 17], an array value in SAC is characterized 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.

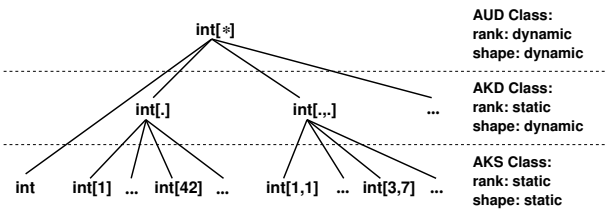


Figure 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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 as structural properties. Both rank and shape vector can be queried by built-in functions. The data

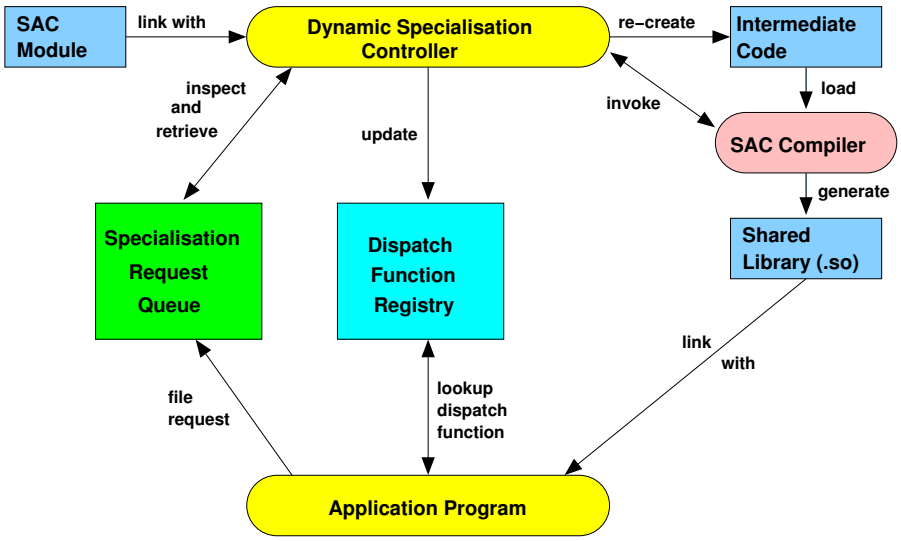


Figure 3. Software architecture of asynchronous adaptive specialization framework

vector can only be accessed element-wise through a selection facility adopting the square bracket notation familiar from other 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.

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, and consequently any shape, 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. For *AUD arrays* both shape vector and rank are dynamic and lead to corresponding runtime data structures.

3. Asynchronous Adaptive Specialization

In order to reconcile software engineering principles for generality with user demands for performance we have developed the asynchronous adaptive specialization framework illustrated in Fig. 3. The idea is to postpone specialization if necessary until runtime, when all structural information is eventually available no matter what. A generic SAC function compiled for runtime specialization 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.

When executed, the proxy function files a specialization 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 specialization has been built before, or whether an identical request is currently pending. In the former case, the proxy function dispatches to the previously specialized code, in the latter case to the generic code, but without filing another request.

Concurrent with the running application, a specialization controller (thread) takes care of specialization requests. It runs the fully-fledged SAC compiler with some hidden command line arguments that describe the function to be specialized and the specialization 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 optimization level and generate a new dynamic library containing the specialized code and a new proxy function. Eventually, the specialization controller links the application with that library and replaces the proxy function in the running application.

The effectiveness of asynchronous adaptive specialization depends on how often the dynamically specialized 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 specialized 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 specialized 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 analyzing an application's properties and by stopping the creation of further specialized 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

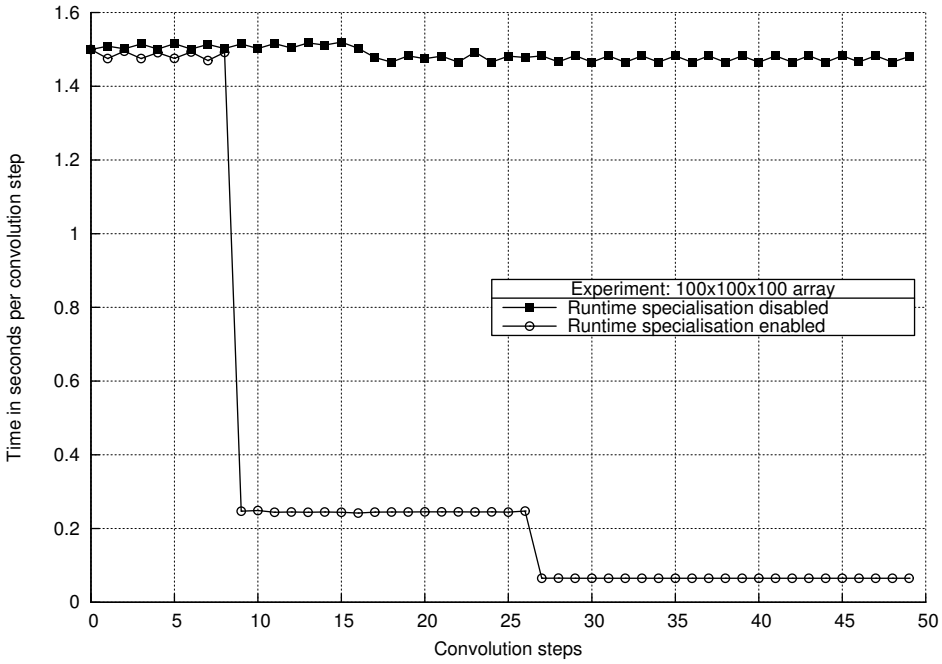


Figure 4. Case study: running a generic convolution kernel on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialization

and compile time of any code are unrelated with each other and, thus, many scenarios are possible.

In order to demonstrate the possible dynamic behaviour of asynchronous adaptive specialization and its impact on application performance, we show the measurements from one experiment in Fig. 4. The experiment was performed 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, but we expect very similar results on different processor architectures.

The experiment is based on a rank-generic convolution kernel with convergence test. In this code two functions are run alternately for a sequence of steps: a convolution step that computes a new array from an existing one and a convergence test that checks whether the old and the new array are sufficiently similar to stop computing. Both functions are defined in rank-generic manner and appropriate measures are put in place to prevent the SAC compiler from statically optimizing either function.

Fig. 4 shows the dynamic behaviour of an application that applies this convolution kernel to a 3-dimensional array of $100 \times 100 \times 100$ double precision floating point numbers. The plot 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 specialization disabled and enabled, respectively.

With asynchronous adaptive specialization disabled the time it takes to complete one cycle consisting of convolution step and convergence check — as expected — is more or less constant. With asynchronous adaptive specialization enabled, however, we can observe two significant drops in per iteration execution time. After 8 iterations running completely generic binary code a shape-specialized version of the convolution step becomes available. Switching from a generic to a non-generic implementation reduces the execution time per iteration from about 1.5 seconds to roughly 0.25 seconds. After 26 iterations in addition to the specialized convolution step also a specialized convergence check has been compiled and linked into the running application. This reduces the execution time of a single iteration further from 0.25 seconds to 0.065 seconds.

This example demonstrates the tremendous effect that runtime specialization can have on generic array code. The main reason for this considerable performance improvement again is the effectiveness of optimizations that fuse consecutive array operations and, thus, avoid the creation of intermediate arrays. A more detailed explanation of this experiment as well as a number of further experiments can be found in [12] and in [10]. All these experiments unanimously substantiate the relevance of asynchronous adaptive specialization in practice.

4. Issue 1: specialization vs overloading

Our first issue originates from SAC's support for function overloading in conjunction with our desire to share specializations between independent applications. The combination of overloading and specialization raises the question how to correctly dispatch function applications between different function definitions of the same name. In Fig. 5 we show an example of 5 overloaded definitions of the function `foo`. The actual bodies of the function definitions are irrelevant in our context and, thus, we leave them out. Moreover, SAC is currently restricted to be monomorphic on the element type of arrays. Hence, we uniformly use type `int` throughout the example.

From a given set of overloaded function definitions the SAC compiler derives explicit dispatch code that dispatches on parameter types from left to right and for each parameter first on rank and then on type. The type system of SAC ensures that the dispatch is unambiguous [19]. More precisely, if the first parameter type of some function instance is a subtype of the first parameter type of some other overloaded instance of the same function, then the same relationship must hold for all further parameter types, etc.

```
1  int[*] foo( int[*]    a, int[*]   b) {...}
2  int[*] foo( int[.]    a, int[.]   b) {...}
3  int[*] foo( int[7]    a, int[8]   b) {...}
4  int[*] foo( int[.,.] a, int[42] b) {...}
5  int[*] foo( int[2,2] a, int[99] b) {...}
```

Figure 5. Example of shapely function overloading in SAC

Fig. 6 shows an excerpt of the wrapper code derived from the original overloading example.

For the construction of the dispatch tree it is irrelevant whether a some instance of a function definition is original user-supplied code or a compiler-generated specialization. There is, however, a significant semantic difference between the two cases that makes our life difficult as it comes to the proposed persistence layer: when dispatching between compiler-generated specializations of the some original function, it is desirable to dispatch to the most specific instance because that is arguably the most efficient one, but it is, not necessary from a correctness point of view. In contrast, when dispatching between different overloaded instances of some function, the compiler must dispatch any application to the best matching instance, no matter what.

With this in mind the obvious question is how we dispatch function applications in the case of the original asynchronous adaptive specialization framework. In fact, we can exploit an interesting feature of the SAC module system. It allows us to import possibly overloaded instances of some function and to again overload those instances with further instances in the importing module. This feature allows us to incrementally add further instances to a function, and this feature is extremely handy when it comes to implementing runtime specialization.

On every module level that adds further instances a new dispatch (wrapper) function similar to that shown in Fig. 6 is generated that implements the dispatch over all visible instances of a function regardless of where exactly these instances are actually defined. We take advantage of this design for implementing asynchronous adaptive specialization as follows: each time we generate a new specialization at application runtime we effectively construct a new module that imports all existing instances of the to be specialized function and then adds one more specialization to the module, the one matching the current function application. Without further ado the SAC compiler in addition to the new executable function instance also generates a new dispatch wrapper function that dispatches over all previously existing instances plus the one newly generated instance. All we need to do at runtime then is to appropriately replace the previously existing dispatch function by the new one.

At first glance, it seems we could continue with this scheme, and whenever we add a further specialization to the repository of specializations we replace the previous dispatch function by the new one. In other words, we would carry over the concept from a single application run to the set of all application runs in the history of the computing system installation.

Unfortunately, this would be incorrect.

The show-stopper here is the coexistence of semantically equivalent specializations and possibly semantically different overloadings of function instances. One dispatch function in the specialization repository is not good enough because any program (or module) may well contribute further overloadings to whatever function definition is available. This may shadow certain specializations in the repository and at the same time require the generation of new specializations that are semantically different from the ones in the repository, despite sharing the same function name.

```
1  int[*] foo_wrapper(int[*] a, int[*] b)
2  {
3    if (dim(a) == 1) {
4      if (shape(a) == [7]) {
5        if (dim(b) == 1) {
6          if (shape(b) == [8]) {
7            c = foo_3( a, b);
8          }
9          else {
10           c = foo_2( a, b);
11         }
12       }
13       else {
14         c = foo_1( a, b);
15       }
16     }
17     else {
18       if (dim(b) == 1) {
19         c = foo_2( a, b);
20       }
21       else {
22         c = foo_1( a, b);
23       }
24     }
25   }
26   else if (dim(a) == 2) {
27     if (shape(a) == [2,2]) {
28       if (dim(b) == 1) {
29         if (shape(b) == [99]) {
30           c = foo_5( a, b);
31         }
32         else if (shape(b) == [42]) {
33           c = foo_4( a, b);
34         }
35         else {
36           c = foo_1( a, b);
37         }
38       }
39       else {
40         c = foo_1( a, b);
41       }
42     }
43     else {
44       if (shape(b) == [42]) {
45         c = foo_4( a, b);
46       }
47       else {
48         c = foo_1( a, b);
49       }
50     }
51   }
52   else {
53     c = foo_1( a, b);
54   }
55
56   return c;
57 }
```

Figure 6. SAC wrapper function with dispatch code for the five overloaded instances of function `foo` shown in Fig. 5

A simple example illustrates the issue: let us assume a function `foo` with, for simplicity, a single argument of type `int[*]`. Again the element type, here `int`, is irrelevant. Let us further assume that the original definition of `foo` is found in module `A`. Now, some application using module `A` may have created specializations for shapes `[42]`, `[42,42]` and `[42,42,42]`, i.e. for 1-dimensional, 2-dimensional and 3-dimensional arrays of size 42 in each dimension.

In this context we write another module B that imports the original definition of function foo, i.e. the generic one, and adds one more instance: foo(int[.,.]). This new instance of foo may not be semantically equivalent to the generic function imported from module A. Of course, it would be good software engineering practice if both function instances that bear the same name are somewhat related, but firstly this cannot be enforced in any way and secondly there may be a good reason to provide the specific definition of function foo for matrices, although it does not yield the exact (bit-wise) same result as applying the original rank-generic definition to a matrix.

As a consequence of the scenario sketched out above, an application of function foo to a vector of 42 elements in module B could be dispatched to the specialized instance in the repository, same as in module A. However, an application of function foo to a matrix of 42x42 elements in module B must be dispatched to the shape-generic instance defined in module B itself. This should trigger a further runtime specialization during the execution of module B. As a consequence, two different instances of function foo both specialized for 42x42 element matrices materialize in the specialization repository.

This raises questions pertaining to the organization of the specialization repository that we elaborate on in Section 5 while we focus on the dispatch issue for now. From the above scenario it becomes clear that we need a two-level dispatch for the persistence layer. Firstly, we must dispatch within the current application. This can be done with the conventional dispatch wrapper functions as illustrated in Fig. 6. If as the result of this first level dispatch a rank- or shape-generic function instance is selected, we must interfere.

First, we focus our attention on the specialization repository. We must figure out whether or not a suitable specialization already exists. For this purpose module name, function name and the sequence of argument types with full shape information (as is always available at application runtime) suffice to identify the correct instance. If the required specialization does already exist, we can directly link it into the running application and call it. If the required specialization does not yet exist, we file the corresponding specialization request, as described in Section 3. Then we call the generic function instance. Asynchronously, the specialization controller will create an executable specialization of this specific generic function instance and likewise asynchronously will add it to the specialization repository when finished with compilation.

5. Issue 2: file system as specialization data base

So far, we have silently assumed some form of specialization collection or data base that allows us to store and retrieve function specializations in a space and time efficient way. To be more concrete now, we deem the file system to be the best option to serve as this persistent data base.

To avoid issues with write privileges in shared file systems we refrain from sharing specilazations between multiple users. While it would appear attractive to do so in particular for functions from the usually centrally stored SAC standard library from a purely technical perspective, the system administration concerns of running SAC applications in privileged mode can hardly be overcome in practice. Consequently, we store specialized function instances in the user's file system space. A subdirectory .sac2c in the user's home directory appears to be a suitable default location.

Each specialized function instance is stored in a separate dynamic library. In order to store and later retrieve specializations we make reuse of an already existing feature within the SAC compiler: to disambiguate overloaded function instances (and likewise compiler-generated specializations) in compiled code we employ a scheme that constructs a unique function name out of module name, function name and argument type specifications. We use that very same scheme, but replace the original separator token (underscore-underscore) by a slash. As a consequence, we end up with a potentially very complex directory structure that effectively implements a search tree and thus allows us to efficiently locate existing specializations as well as to identify missing specializations.

There is, however, one small pitfall that luckily can be overcome fairly easily. A module name in SAC is not necessarily unique in a file system. Like many other compilers the SAC compiler allows users to specify directory paths to locate modules in the file system. Changing the path specification from one compiler run to the next may effect the semantics of a program. Like with any other compiler, it is the user's responsibility to get it right. For our purpose this merely means that instead of the pure module name we need to use a fully qualified path name to uniquely identify a module definition.

6. Issue 3: semantic revision control

For users who merely run SAC application programs instead of writing their own the techniques described in the preceding two sections would be sufficient. Of course, forbidding users to write their own SAC code when making use of persistent asynchronous adaptive specialization is a fairly undesirable constraint.

So, what is the issue?

Let us go back to the scenario sketched out in Section 4. The user's specialization repository contains four specializations, three specializations of the function foo(int[*]) as defined in module A ([42], [42,42] and [42,42,42]) and one specialization of function foo(int[.,.]) as defined in module B (again [42,42]).

A developing user could now simply come up with the idea to change the implementation of function foo(int[.,.]) in module B and by doing so invalidate certain existing specializations in the repository. To be on the safe side, we must incorporate the entire definition of a rank- or shape-generic function into the identifier of a specialization.

For this purpose we linearize the intermediate code of a generic function instance into textual form and compute a suitable hash when generating a dynamic specialization of this generic instance. This hash is then used as the lowest directory level when storing a new specialization in the file system.

Upon retrieving a specialization from the file system repository a running application again generates a hash of a linearization of the intermediate code of its own generic definition and uses this for generating the path name to look up the existence of a specific specialization needed.

With this non-trivial solution we ensure that we never accidentally run an outdated specialization.

7. Issue 4: specialization repository size control

A rather obvious issue in persistent asynchronous adaptive specialization is the need to control the size of a specialization repository in some suitable way. Otherwise, the scheme as described so far is bound to accumulate more and more specializations over time. With today's typical disk spaces this is not an immediate problem, but of course it will become one over time, no matter what. Requiring the user to manually discard all specializations when running out of disk space is not an attractive solution.

Instead, we ask the user at installation time how much disk space he would like to give SAC for the specialization repository; of course, this could be changed later. Now, the specialization repository becomes a sot of cache memory. As long as the size limit has not been reached, we simply let it grow. When the size limit is reached, we must create space before storing a new specialization. As is common in cache organizations, we expect the least recently used specialization across all modules, functions, etc. to be the least

likely to be used in the future. This is of course just a heuristics, but it has worked reasonably well in hardware caches and in the absence of accurate prediction of the future there is not much we could do to be much smarter. Given the heuristic nature of this approach we can — just as hardware caches do — get away with a reasonable approximation of the least recently used property.

File system time stamps provide all the necessary information for free. Unfortunately, searching for the file with the oldest access time stamp in a reasonably large specialization repository can be unpleasantly time consuming. Of course, this would happen asynchronously to the running application in a specialization controller thread, but notwithstanding it makes sense to think about a smarter scheme.

Our plan is to store a small file containing the access time stamp of the least recently accessed file in the whole directory. Originally, this is the creation time of the first file in some directory (and that of the directory). Adding a new file (or subdirectory) to a directory does not affect this time stamp because that file would have a newer time stamp.

However, if a specialization is loaded from the repository the access time stamp of the corresponding file is updated. If that file is/was the oldest in the repository (i.e. its time stamp coincides with that stored in the special file), the time stamp in the special file will be updated to the now oldest time stamp found in the directory. If so, we go one directory up and check if the special file on that level contains exactly the given time stamp. If so, we must update the information of this directory level. Since directory time stamps do not accurately reflect accesses to subdirectories, we must rely on our own time stamp files. In this case we search for the special file with the oldest time stamp among all subdirectories and copy this file (or rather its contents) into the current directory. We recursively repeat this procedure until we reach the top level of the specialization repository.

If new a specialization is to be stored in an already full specialization repository, we can now efficiently locate the least recently accessed specialization in the whole repository by going top-down from the root of the directory tree always choosing the least recently accessed subdirectory based on the time stamps in the special files. After deleting the least recently used specialization, we recursively go up the directory tree again applying the exact same technique as described above for loading a specialization.

The advantage of the proposed scheme is that its overhead is linear in the depth of the tree, not in the size of the tree as a naive search. The scheme is, nonetheless, not fully accurate as it only recognizes when a specialization is loaded into a running application, not how often that specialization is effectively used in that application. One can think of a refinement that updates the access time stamps above whenever a specialized function instance from the repository is actually executed within an application. It is, however, not a-priori clear that the additional overhead that such a refinement would bring with it on average pays off. We consider this an area of future research to give more substantiated answers to these questions.

8. Conclusions

Asynchronous adaptive specialization 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 reality whenever application programmer and application user are not identical, which rather is the norm than the exception in (professional) software engineering.

In the past we proposed several improvements and extensions to asynchronous adaptive specialization that generally broaden its applicability by making specialized binary code available quicker [10]. One key proposal was to make specializations persistent. Persistent asynchronous adaptive specialization aims at sharing runtime overhead across several runs of the same application or even across multiple independent applications sharing the same core library code (e.g. from the SAC standard library).

In the ideal case required specializations of some function do not need to be generated on demand in a time- and resource-consuming way at all. Instead, following some learning or setup period the vast majority of required specializations have already been generated in preceding runs of the same application or even other applications that share some of the code base, as for example parts of SAC's comprehensive standard library. If so, these pre-generated specializations merely need to be loaded from a specialization repository and linked into the running application. In many situations the proposed persistence layer may effectively reduce the average overhead of asynchronous adaptive specialization to close to nothing.

What appeared to be very attractive but mainly an engineering task at first glance has proven to be fairly tricky in practice. In this paper we identified a number of issues related to correct function dispatch in the presence of specialization and overloading, use of the file system as code data base, revision control in the potential presence of semantically different function definitions and, last not least, control of the specialization repository size. We sketched out our solutions found for each of the four issues.

Currently, we are busy implementing the various proposed solutions. In the near future we expect to run experiments that demonstrate how we can reconcile abstract specifications with high sequential and parallel execution performance, seemingly without observable overhead.

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.

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

[3] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany, 2001. Logos Verlag, Berlin, 2001.

[4] 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.

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

[6] C. Grelck and S.-B. Scholz. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In N. Glew and G. Blelloch, editors, *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*, pages 25–33. ACM Press, 2007.

[7] 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.

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

[9] 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.

[10] C. Grelck and H. Wiesinger. Next generation asynchronous adaptive specialization for data-parallel functional array processing in sac. In R. Plasmeijer, editor, *Implementation and Application of Functional Languages, 25th International Symposium, IFL 2013, Nijmegen, Netherlands, Revised Selected Papers*. ACM, 2014.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

[18] 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.

[19] S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.