International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Persistent Asynchronous Adaptive Specialization for Generic Array Programming

Clemens Grelck · Heinrich Wiesinger

the date of receipt and acceptance should be inserted later

Abstract Generic array programming systematically abstracts from structural array properties such as shape and rank. As usual, generic programming comes at the price of lower runtime performance. The idea of asynchronous adaptive specialization is to exploit parallel computing facilities to reconcile these conflicting objectives through the continuous adaptation of running applications to the ranks and shapes of their arrays.

A key parameter for the effectiveness of our approach is the time it takes from requesting a certain specialization until its availability to the running application. We describe the ins and outs of a persistence layer that keeps specialized variants in a repository for future use and thus effectively reduces the average waiting time for re-compilation to nearly zero. A number of critical issues that, among others, stem from the interplay between function specialization and function overloading catch our special attention. We describe the solutions adopted and illustrate the benefits of persistent asynchronous adaptive specialization by a series of experiments.

1 Introduction

Software engineering is concerned with the fundamental trade-off between abstraction and performance. In array programming this trade-off is between abstracting from ranks and shapes of arrays in source code and the ability to determine actual ranks and shapes through compilation technology as a prerequisite for high runtime performance. However, concrete rank and shape information as a matter of fact is regularly not available before application runtime. For example, data may be read from a file or may be produced by external library code. In these cases the effect of compile time specialization is very limited.

University of Amsterdam, Netherlands
E-mail: c.grelck@uva.nl   h.m.wiesinger@student.uva.nl

Such scenarios motivate our current research in the context of the purely functional, data-parallel array language SaC (Single Assignment C) [1,2]. 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 likewise from their ranks (i.e. number of axes).

In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and even more so for rank-generic code. Apart from the obvious reason that generic code maintains more information in runtime data structures, the crucial issue are the SaC compiler's advanced optimizations [3] that are not as effective on generic code as they are on shape-specific code. This is a matter of principle and not owed to implementation deficiencies. For example, in automatically parallelized code [4–6] many organizational decisions must be postponed until runtime, and synchronization and communication overhead is generally higher.

Our asynchronous adaptive specialization framework [7] builds on today's ubiquity of multi-core processor architectures. Asynchronous with the execution of generic code, be it sequential or automatically parallelized, a *specialization controller* generates an appropriately specialized and highly optimized binary clone of some generic function, all while the application continues running the original generic code. Eligible functions are indirectly dispatched such that if the same function is repeatedly called with arguments of the same shapes, its fast clone is used as soon as it becomes available. All this is entirely compiler-directed and completely transparent to the programmer/user.

The effectiveness of our approach critically depends on making specialized binary clones available as quickly as possible. This would normally call for a fast and light-weight just-in-time compiler. Unfortunately, the SaC compiler is everything but light-weight. Rewriting it in a more light-weight style would not only incur a gigantic engineering effort, but making the compiler faster would inevitably come at the expense of reducing its aggressive optimization capabilities. Obviously, this would be adverse to our central goal of high performance.

In our original approach [7] specializations are accumulated during one execution of an application and are automatically removed upon the application's termination. Consequently, any subsequent run of the same application starts specializing again from scratch. Of course, the next run may use arrays of different ranks and shapes, but in many real world scenarios it is quite likely that a similar set of shapes will prevail again. The same holds across different applications since SaC programs make heavy use of our comprehensive standard library of rank-generic array operations.

We first proposed the idea of a persistence layer that could in practice reduce the overhead to near-zero in [8]. For many application scenarios we would envision a training phase, after which most required specializations have been generated, and the whole dynamic specialization machinery would only become active again when the user re-runs an application on array shapes not previously encountered.

As a concrete scenario consider an image filter pipeline. Image filters can typically be applied to images of any size. In practice, however, a fairly small number of different image formats prevails, i.e. those produced by the available cameras. Still, the concrete image formats are unknown at compile time of the image processing application. So, our approach would effectively train the application to the image formats of interest. Purchasing a new camera may introduce new image formats and thus would lead to a short re-training phase.

In fact, the proposed persistence layer requires more radical changes to the dynamic specialization framework than initially anticipated. The contributions of this paper are

- to describe *persistent* asynchronous adaptive specialization in detail;
- to identify and solve a number of non-trivial technical issues;
- to illustrate the performance gains achieved by the persistence layer through a series of experiments.

The remainder of the paper is organized as follows. Sections 2 and 3 provide background information on SaC and its runtime specialization framework, respectively. We describe the proposed persistence layer in Section 4. The following two Sections 5 and 6 deal with the particular challenges we encountered, before Section 7 reports on our experimental evaluation. Finally, we sketch out some related work in Section 8 and draw conclusions in Section 9.

2 SAC — Single Assignment C

As the name suggests, SaC leaves the beaten track of functional languages and adopts a C-like syntax: we interpret assignment sequences as nested let-expressions, branching constructs as conditional expressions and loops as syntactic sugar for tail-recursive functions. For any details regarding language design we refer the interested reader to [1,2].

Following the example of interpreted array languages, such as Apl or J, an array value in SaC is characterized by a triple $(r, \mathbf{s}, \mathbf{d})$. The *rank* $r \in \mathbb{N}$ defines the number of dimensions (or axes) of the array. The *shape vector* $\mathbf{s} \in \mathbb{N}^r$ yields the number of elements along each of the r dimensions. The *data vector* $\mathbf{d} \in T^{\prod \mathbf{s}}$ contains the array elements, the so-called *ravel*, where T denotes the element type. Rank and shape vector can be queried by built-in functions. Selection of array elements is done through index vectors. This is owed to the support for rank-generic code that requires an indexing facilities for statically unknown numbers of dimensions and thus indices.

The type system of SaC is monomorphic in the element type but polymorphic in the rank and shape of arrays. As illustrated in Fig. 1, each type induces a three-layer array type hierarchy. On the lowest level we find non-generic types that define arrays of fixed shape. On the intermediate level we see arrays of fixed rank. And, on the top of the hierarchy we find arrays of any rank (and thus any shape). The hierarchy of array types induces a subtype relationship (with function overloading) and leads to three different runtime representations of arrays.

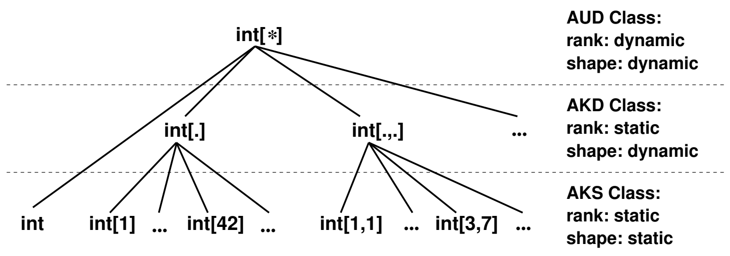

Fig. 1 Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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 statically unknown and, thus, must dynamically be maintained through appropriate runtime data structures. The absence of static rank information here means that the length of the shape vector is not known and, therefore, the shape vector must dynamically be allocated and de-allocated, which incurs relevant cost.

```
import ConvolutionAuxiliaries: {convolution_step, is_convergent};

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

  return A;
}
```

Fig. 2 Rank-generic convolution kernel with convergence test in SaC

We illustrate the use of SaC by the rank-generic convolution kernel shown in Fig. 2, which will serve as a running example throughout this paper. Following the import of functions `convolution_step` and `is_convergent` from the module `ConvolutionAuxiliaries`, we define the function `convolution` that is parameterized over the to be convolved array of any rank and shape `A` and the convergence threshold `eps`; it yields the convolved array. The `do-while` loop is syntactic sugar for tail-recursion. Note that the statement `A_old = A;` does not copy the array, as for instance in superficially similar Fortran-90 code, but merely creates a new lambda-binding to an existing (array) value. For a complete discussion of the example we refer the interested reader to [8], for a thorough introduction of SaC to [2]

3 Asynchronous Adaptive Specialization

The idea of asynchronous adaptive specialization is to postpone specialization if necessary until application runtime, when structural information is always available. A generic SaC function is compiled into two binary functions: the usual generic implementation and a small proxy function that is actually called from outside. When executed, the proxy function first checks whether a previously specialized function instance for the concrete argument ranks and shapes already exists. If so, it dispatches to that fast clone. Otherwise, the proxy function files a specialization request consisting of the function identifier and the concrete argument shapes before calling the generic implementation.

Concurrent with the running application, specialization controllers take care of specialization requests. They run the fully-fledged SaC compiler on an intermediate representation of the function to be specialized and the corresponding specialization parameters. Eventually, they link the resulting binary code into the running application and update the proxy function accordingly.

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. First, the application itself must apply an eligible function repeatedly to arguments with the same shape. Second, 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 this 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 specializations 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, consequently, many scenarios are possible.

We illustrate the effect of asynchronous adaptive specialization by the experiment shown in Fig. 3. This experiment is based on a rank-generic convolution kernel with convergence test, as shown in Fig. 2. In this code two functions are run alternately for a number of iterations: 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 the iteration. Both functions are defined in rank-generic style and appropriate measures are put in place to prevent the SaC compiler from statically specializing either function.

Fig. 3 shows the dynamic behaviour of this rank-generic convolution kernel when applied to a 3-dimensional array of $100 \times 100 \times 100$ double precision

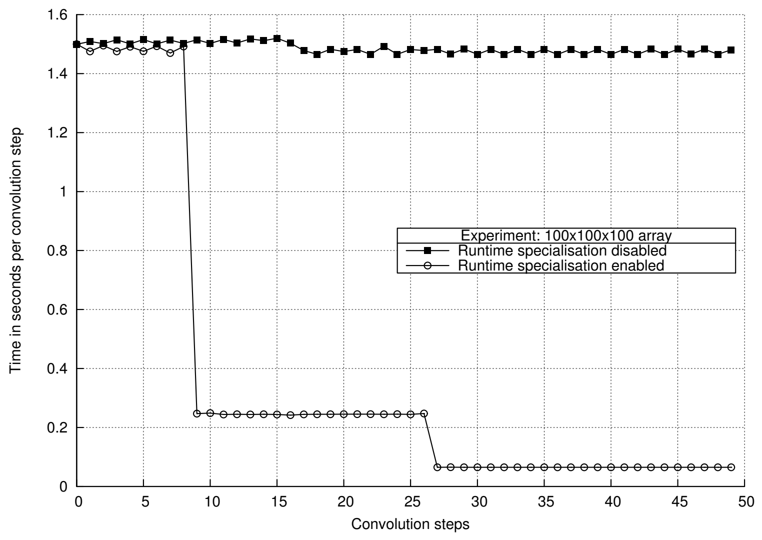

Fig. 3 Experiment: rank-generic convolution kernel on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialization (reproduced from [7])

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. One can easily identify two steps of performance improvement when the specialized variants of the convolution step and the convergence test successively become available to the running application.

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 [7] and in [8].

4 Persistent Asynchronous Adaptive Specialization

The idea of a persistence layer is as intriguing as simple, the latter at least at first glance: Instead of discarding all generated specializations upon termination of each execution of some program, we keep them in a repository for later use by the same application or even by different applications.

Persistent dynamic specialization is a win-only approach. If a required specialization has already been generated by a previous run of the same application or likewise by a previous run of some other application, it can be linked into the running application without any delay, and the costly dynamic compilation process is entirely avoided. This scenario not only makes the fast non-generic clone of some function immediately available to the running application, but also saves the hardware that would otherwise be utilized for recompilation. This either saves energy through partial shut-down of computing resources or makes more resources available to the parallel execution of the application itself thus resulting in higher execution performance.

The file system is the place to keep specialization repositories. To avoid issues with write privileges in shared file systems we refrain from sharing specializations between multiple users. While it would appear attractive to do so, in particular for functions from the usually centrally stored SaC standard library, 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 reuse an already existing mechanism in 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 adapt this scheme by replacing the original separator token by a slash. As a consequence, we end up with a potentially complex directory structure that effectively implements a search tree and thus allows us to locate existing specializations and to identify missing specializations equally efficiently.

There is, however, one pitfall: 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. Changing the path specification may effect the semantics of a program. For our purpose this means that instead of the pure module name we need to use a fully qualified path name to uniquely identify a module definition.

Fig. 4 illustrates the proposed solution with a small example file system layout based on the implementation of the rank-generic convolution kernel with convergence test introduced in Section 2. In the example of Fig. 4 we can easily identify two variants of the module `ConvolutionAuxiliaries`, one in a subdirectory `mylibs` and the other in a subdirectory `myalternativelibs`. Which of these two implementations of the module would be used by a SaC application solely depends on the SaC compiler's path configurations and the command line arguments given and, thus, are orthogonal to the semantics of the language.

In the file system snapshot of Fig. 4 we can further observe the base type encoding of the functions `convolution_step` and `is_convergent`. In the code sketched out in Fig. 2 they are defined for one double precision floating point argument and for three double precision floating point arguments, respectively. To illustrate the potential of different overloaded argument base types we also consider a version of the convergence check for single precision floating point numbers.

```
$HOME
└── .sac2c
    └── rtspec
        └── sac
            ├── mylibs
            │   └── ConvolutionAuxiliaries
            │       ├── convolution_step
            │       │   └── double
            │       │       ├── 2-1000-1000.so
            │       │       └── 3-100-100-100.so
            │       └── is_convergent
            │           ├── double-double-double
            │           │   ├── 2-1000-1000-2-1000-1000-0.so
            │           │   └── 3-100-100-100-3-100-100-100-0.so
            │           └── float-float-float
            └── myalternativelibs
                └── ConvolutionAuxiliaries
                    ├── convolution_step
                    │   └── double
                    │       ├── 1-1000000.so
                    │       └── 3-100-100-100.so
                    └── is_convergent
                        ├── double-double-double
                        │   ├── 1-1000000-1-1000000-0.so
                        │   └── 3-100-100-100-3-100-100-100-0.so
                        └── float-float-float
```

Fig. 4 Example file system layout with multiple variants of a SaC module

On the level of individual dynamic libraries we can see the encoding of the exact ranks and shapes of existing dynamic specializations: for the first implementation of our module we can see two specializations, one for a 2-dimensional case (1000x1000) and one for a 3-dimensional case (100x100x100). For the alternative implementation we can identify a 1-dimensional specialization (1,000,000) and again the same 3-dimensional specialization as before. The third argument of our function `is_convergent` is a `double` scalar, the convergence threshold. This incurs the trailing zero in all corresponding file names: the rank of the 3rd argument.

For the time being we prefer clarity and readability over obfuscation and intellectual property protection. The literal encoding of argument types and shapes can easily be replaced by one that discloses less information about the (source) code in line with user demands.

5 Persistent Specialization vs Function Overloading

As usual, the devil is in the detail, and so we discovered a number of issues that make the actual implementation of persistent asynchronous adaptive specialization much more challenging than originally anticipated. Our first issue originates from SaC's support for function overloading in conjunction with our aim to share specializations between multiple applications. The combination

```
int[*] foo( int[*]   a, int[*]  b);
int[*] foo( int[.]   a, int[.]  b);
int[*] foo( int[7]   a, int[8]  b);
int[*] foo( int[.,.] a, int[42] b);
int[*] foo( int[2,2] a, int[99] b);

int[*] foo_dispatch( int[*] a,
                     int[*] b)
{
  if (dim(a) == 1) {
    if (shape(a) == [7]) {
      if (dim(b) == 1) {
        if (shape(b) == [8]) {
          c = foo_3( a, b);
        }
        else {
          c = foo_2( a, b);
        }
      }
      else {
        c = foo_1( a, b);
      }
    }
    else {
      if (dim(b) == 1) {
        c = foo_2( a, b);
      }
      else {
        c = foo_1( a, b);
      }
    }
  }

  ...
```

```
  ...

  else if (dim(a) == 2) {
    if (shape(a) == [2,2]) {
      if (dim(b) == 1) {
        if (shape(b) == [99]) {
          c = foo_5( a, b);
        }
        else if (shape(b) == [42]) {
          c = foo_4( a, b);
        }
        else {
          c = foo_1( a, b);
        }
      }
      else {
        c = foo_1( a, b);
      }
    }
    else {
      if (shape(b) == [42]) {
        c = foo_4( a, b);
      }
      else {
        c = foo_1( a, b);
      }
    }
  }
  else {
    c = foo_1( a, b);
  }

  return c;
}
```

Fig. 5 Example of shapely function overloading and resulting dispatch function

of overloading and specialization raises the question how to correctly dispatch function applications between different function definitions bearing the same name. In Fig. 5 we show an example of five overloaded definitions of the function `foo` alongside the compiler-generated dispatch code. We dispatch on parameter types from left to right and for each parameter first on rank and then on shape. The type system ensures that the dispatch is unambiguous.

For the construction of the dispatch tree it is irrelevant whether some instance of a function is original code or a compiler-generated specialization. There is, however, a significant semantic difference: while we aim at dispatching to the most specific compiler-generated specialization for performance reasons, we must dispatch to the best matching user-defined instance no matter what. To achieve this our original asynchronous adaptive specialization framework exploits an interesting feature of our module system, which 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.

On every module level that adds further instances we generate a new dispatch function similar to that in Fig. 5. This implements the dispatch over all visible instances 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 function that dispatches over all previously existing instances plus the newly generated instance. All we need to do at runtime then is to replace the old dispatch function by the new one.

At first glance, it looks as if we could continue with this scheme, and whenever we add a specialization to the repository, we simply replace the dispatch function in the repository by the new one. Unfortunately, carrying over this concept from a single application run to all application runs in the history of the computing system installation would violate overloading semantics.

The issue 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. This may semantically 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.

A simple example illustrates the issue: let us assume a module `A` that exports a function `foo` with, for simplicity, a single argument of type `int[*]`. Again, the element type `int` is irrelevant. Now, some application(s) using module `A` may have created specializations in the repository for shapes `[42]`, `[42,42]` and `[42,42,42]`. One may think that the repository could also simply contain a dispatch function that dispatches between all repository instances, but this is not an option for several reasons.

Firstly, the repository instances are created incrementally, possibly by multiple applications using `A::foo`. New dispatch functions could only be created during a dynamic compilation process. For that the compiler would need to know all existing specializations in the repository in order to create the new dispatch function for these and the one new specialization. Trouble is that multiple dynamic compilations may happen simultaneously, which immediately creates a mutual exclusion problem on the repository and would require a lock for the repository. This lock would be needed throughout the whole dynamic compilation process. Thus, significant delays in completing asynchronous adaptive specializations could be expected.

Secondly, with substantial repository sizes, dynamic dispatch over many instances in the style of Fig. 5 becomes increasingly inefficient because most applications may effectively only ever make use of a small fraction of the instances accumulated in the repository after some time.

Thirdly, imagine a program that itself has an overloading of function `foo` for 42-element vectors. This program would have to internally dispatch to its own instance of `foo(int[42])`. At the same time it could still make beneficial use the repository instances of `foo(int[42,42])` and `foo(int[42,42,42])`, and it may even create a new repository instance for, say, `foo([int[21])`.

From the above scenarios it becomes clear that for the persistence layer we need a two-level dispatch. First, we dispatch within the running application through a conventional dispatch function, as illustrated in Fig. 5. If this dispatches to a rank- or shape-generic instance, we interfere and determine whether or not a suitable specialization already exists in the specialization repository. For this purpose module name, function name and the sequence of argument types with full shape information (as always available at application runtime) suffice to identify the corresponding shared library in the file system.

If the required specialization does already exist, we directly link this instance into the running application and call it. Now, we need a second-level dispatch mechanism that keeps track of all dynamically linked instances. A classical dispatch function, as used so far, is not an option. We deliberately avoided dynamic compilation to reduce overhead, and so compiling a new dispatch function would be counter-productive. Instead, we use a dynamic data structure to store function pointers for dynamically loaded instances with the function name and parameter types and shapes serving as search keys. For now, we use a simple search tree in our concrete implementation, but this could easily be replaced by a more sophisticated mechanism in the future.

If the required specialization does not yet exist, we file the corresponding specialization request as before and call the generic function instance instead. However, more changes are needed in this case as well. When the dynamic compilation process completes, we do no longer link the new binary version of the additional instance into the running application. After all, it is pure speculation that this application will ever call it. Instead, we create the corresponding shared library in the specialization repository for future use by this and possibly other applications. Should the running application ever need this specific instance (again), it will load the instance from the specialization repository, just as described in the previous paragraph.

6 Semantic Revision Control

Consider once more the scenario of a function `foo(int[*])` and a specialization repository that contains three specializations for shapes `[42]`, `[42,42]` and `[42,42,42]`. Furthermore, let us assume a program overloads `foo` with another generic instance `foo([int[.,.])`. If this program calls `foo` with a 42-element vector, we can load the corresponding previously specialized instance and benefit from high performance. However, if this program calls `foo` with a 42x42-element matrix, we must not load the corresponding instance from the repository because that is derived from `foo(int[*])`. Instead, overloading semantics require us to use the local instance `foo(int[.,.])` as basis for futher specialization. Since that is a generic function as well, we want to use our asynchronous adaptive specialization mechanism once more. That inevitably leads to two non-identical and potentially demantically diverging instances `foo(int[42,42])` in the repository, one derived from `foo(int[*])` and one derived from `foo(int[.,.])`.

```
...
└── mylibs
    └── ConvolutionAuxiliaries
        ├── convolution_step
        │   └── double
        │       ├── 2-1000-1000
        │       │   ├── dcc0e68deb5ffd37f4ab45e30e13c8.so
        │       │   └── f9783fc9338fda9f48f4b79335479e.so
        │       └── 3-100-100-100
        │           ├── dcc0e68deb5ffd37f4ab45e30e13c8.so
        │           └── f9783fc9338fda9f48f4b79335479e.so
        └── is_convergent
            └── double-double-double
                ├── 2-1000-1000-2-1000-1000-0
                │   ├── db60c387568b3665c5b099a8d3d675.so
                │   └── e5219fecc5606c6f3a47c2411efb1c.so
                └── 3-100-100-100-3-100-100-100-0
                    ├── db60c387568b3665c5b099a8d3d675.so
                    └── e5219fecc5606c6f3a47c2411efb1c.so
```

Fig. 6 Refinement of the file system layout shown in Fig. 4 considering multiple semantic variants of the same SAC module, distinguished by unique hashes of the intermediate code representation (shortened to fit horizontally)

This scenario exemplifies a dilemma that has another variant. A developer could simply come up with the idea to change the implementation of function `foo(int[*])` in module `A`. This somewhat invalidates certain existing specializations in the repository, but this invalidation only becomes effective after the application itself is recompiled. Consequently, we face the situation where some applications "see" different specializations of the same function for identical type and shape than other applications.

To solve both issues at once we need a mechanism that keeps track of what exact generic code any instance in the repository is derived from. Therefore, we must incorporate the entire definition of a generic function into the identifier of a specialization. For this purpose we linearize the intermediate code of a generic function definition into textual form and compute a suitable hash when generating a dynamic specialization. This hash is then used as the lowest directory level when storing new specializations 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. This is used to determine whether or not a suitable specialization exists in the repository and to locate the correct one. With this non-trivial solution we effectively ensure that we never accidentally run an outdated specialization.

Fig. 6 illustrates this solution for our running example of the rank-generic convolution kernel. Compared with the file system layout in Fig. 4 the shape encoding has now become yet another level in the file system hierarchy while the actual binary code resides in a collection of dynamic libraries whose names are hash sums of their intermediate representations.

7 Experimental Evaluation

In our experimental evaluation of persistent asynchronous adaptive specialization, we repeat a series of experiments initially reported on in [8]. These involve three different benchmarks: generic multi-dimensional convolution with periodic boundary conditions and convergence test, as illustrated in Fig. 2, repeated matrix multiplication and n-body simulation. The test system for all three case studies is a 48-core SMP machine with 4 AMD Opteron 6172 Magny-Cours processors running at 2.1 GHz and a total of 128 GB of DRAM. Each core has 64 KB L1 instruction cache, 64 KB L1 data cache and 512 KB unified L2 cache. Each group of 6 cores shares a unified L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5. All reported figures are best of five independent runs.

In [8] we explicitly discussed the combination of automatically parallelized applications with multiple concurrent specialization controllers. In the following we restrict ourselves to sequential program execution and a single specialization controller to isolate the effect of the proposed persistence layer, which is the novel contribution of this paper.

One may say that the variants that employ adaptive specialization effectively use more resources than the ones without, more precisely two cores instead of one, and that this constitutes an unfair comparison. We do not subscribe to this point of view for the following reasons. Firstly, on the 48-core machine used we would need to compare using all 48 cores for parallel execution of the application with only using 47 cores for the application and one for asynchronous adaptive specialization. Even with (unlikely) perfect linear scaling of the application, the performance difference between using 47 cores and using 48 cores would be marginal. Secondly, faster execution of the application due to parallelization would indeed change the speed ratio between the application and the compiler. However, this would not be different from increasing or decreasing the problem size, that we rather arbitrarily chose with the purpose of best possible illustration.

7.1 Rank-generic convolution with convergence test

For thorough discussions of our first benchmark regarding both general language design and asynchronous adaptive specialization we refer the interested reader to [2] and to [8], respectively. In essence, the benchmark alternately computes one convolution step and the convergence check. Both functions, hereafter named `step` and `check` for brevity, are defined in a rank-invariant style, i.e. they can be applied to argument arrays of any rank and shape. The `step` function uses a star-shaped neighbourhood. For a 2-dimensional argument array this results in a 5-point stencil, for a 3-dimensional argument array in a 7-point stencil, etc. Fig. 7 shows the outcome of our experiments for a case of 3-dimensional convolution with 100x100x100 double precision floating point numbers. On the x-axis we show 29 iterations and on the y-axis the execution time for each iteration as measured by a high precision clock.

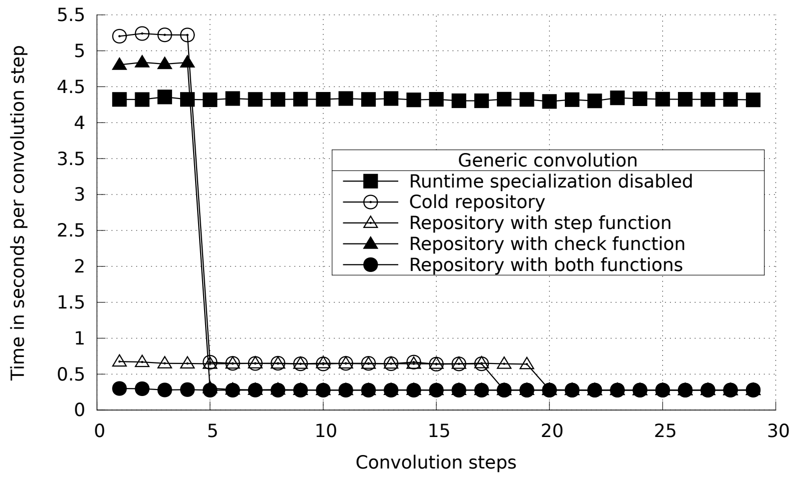

Fig. 7 Performance impact of the persistence layer for the generic convolution kernel with convergence check

Without runtime specialization, i.e. when continuously running fully rank-generic code, each iteration takes about 4.4 seconds. The computation is completely uniform across iterations. Thus, the small variations in the execution time are purely caused by (negligible) operating system and other activities on the system and the corresponding measurement inaccuracies.

With runtime specialization enabled and an empty (or cold) specialization repository the first four iterations take 5.2 seconds each. During this time the concurrently running specialization controller generates an optimized variant of the convolution step function, as this is the first of the two relevant functions to be applied. A shape-specialized version of the convolution, while still running the rank-generic convergence check, brings the average execution time per iteration down to about 0.65 seconds.

After 13 further iterations the specialization controller has additionally generated a shape-specialized version of the convergence check, which brings down the average execution time per iteration to approximately 0.28 seconds. Since the actually running code is now fully shape-specific, no further changes in the execution time can be expected.

The third line in Fig. 7 shows the runtime behaviour if the needed specialization of the convolution step already exists in the specialization repository. If so, per iteration execution time is 0.65 seconds from the first iteration on.

In this scenario we immediately start the specialization of the convergence check, which becomes available after 19 iterations, further reducing the per iteration execution time to the optimal value of 0.28 seconds. Note that the absolute performance is considerably better than in the previous scenario as 19 iterations are much earlier reached than the 17 iterations that led to optimal performance in the previous scenario.

The fourth line in Fig. 7 shows the inverse case where the specialization repository contains the required version of the convergence check but not that of the convolution step. Since the performance impact of the convolution step is far greater than that of the convergence check, we observe a moderate performance improvement for the first four iterations. If both required specializations are already present in repository at program startup, all iterations execute in about 0.28 seconds from the very beginning

These observations can be considered representative. Using different problem sizes changes the ratio between dynamic re-compilation times and application execution times in the foreseeable way. We thus do not report on further problem sizes and refer the interested reader to [7] for a detailed discussion and further experimental data.

7.2 Repeated matrix multiplication

Repeated matrix multiplication is a benchmark that we adopt from [8] as well. Here, we apply a shape-generic matrix multiplication function to two argument matrices of 1000x1000 double precision floating point numbers. Then we repeatedly multiply the resulting temporary matrix with the second argument matrix for a given number of times, see [8] for the complete source code. In this case we have only one function relevant for dynamic specialization: `matmul`. Fig. 8 shows the outcome of our experiment. We essentially observe a similar

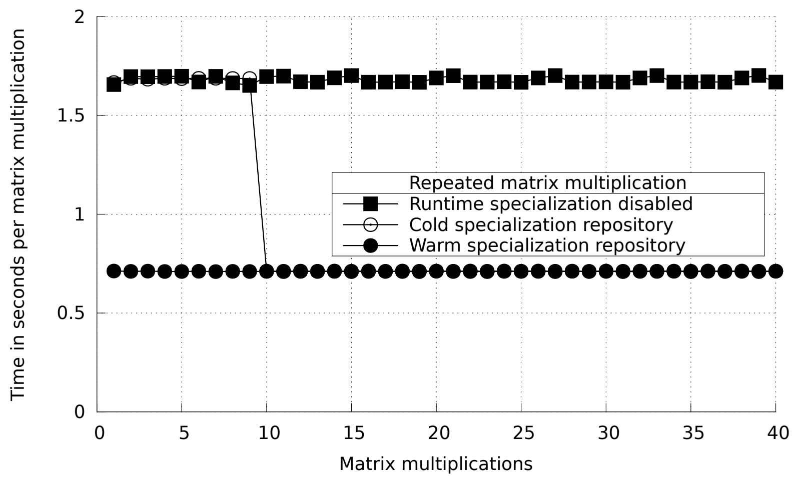

Fig. 8 Impact of the persistence layer on runtime performance for the repeated matrix multiplication benchmark for the concrete problem size 1000x1000

runtime behaviour as in the case of the rank-generic convolution kernel, but only one step of performance improvement.

Comparing our latest findings with Fig. 8 in [8] we can see that the additional overhead due to the persistence layer is below measurement accuracy: We essentially observe the same overhead as in our previous experiments. If the right variant of the `matmul` function can directly be retrieved from the persistent specialization repository, we immediately obtain the best possible performance from the first iteration onwards.

7.3 N-body simulation

N-body simulation is our third benchmark adopted from [8]. A comprehensive account of n-body simulation in SaC can be found in [9]. Here, we again have two different generic functions: `advance` for computing one simulation step and `energy` to assess the overall energy in the simulated system. Compared with the convolution kernel an important difference in code structure is that the `energy` function is called exactly twice: once before and once after the time iteration. In practice, the `energy` function's impact on overall performance depends on the number of time steps simulated and usually becomes irrelevant at some level. Nonetheless, our compiler specializes it. We illustrate the outcome in Fig. 9.

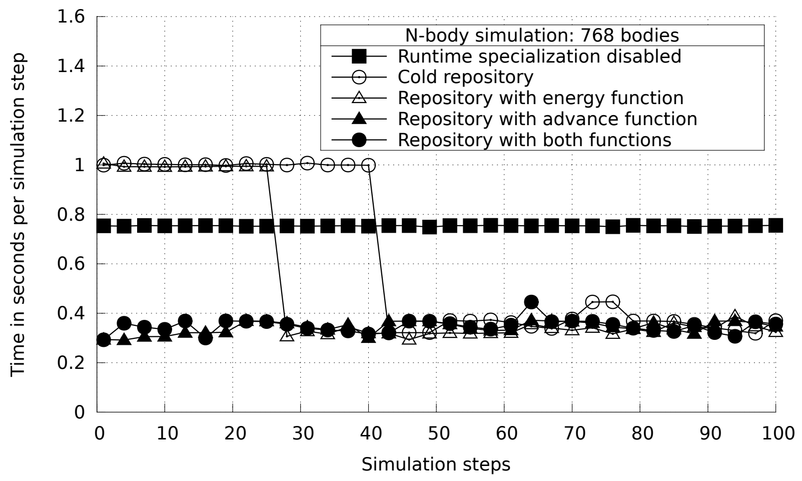

Fig. 9 Performance impact of persistence layer in n-body simulation

Since the `energy` function is the first of the two relevant functions to be applied, it will also be specialized first. This occupies the specialization controller for a substantial amount of time and, thus, delays the much more important runtime specialization of the `advance` function. Although the `energy` function is not really performance-critical, having the right variant in the persistent specialization repository has a disproportional positive performance impact as the specialization controller can now immediately start to generate the appropriate specialization of the more important `advance` function.

In Fig. 10 we look at the same experimental data from a different perspective. We show whole program runtimes for different initial states of the persistent specialization repository as well as for runtime specialization disabled for the same 100 iterations as before. In other words, the numbers refer to the integrals below the lines in Fig. 9. Additionally, we show accumulated runtimes of the specialization controller. Even for a cold specialization repository we achieve better results as without dynamic adaptation. These numbers would quickly move further in favour of our technique as we run more iterations of the n-body simulation. Likewise, we can see that almost all remaining overhead can be avoided by means of the persistence techniques proposed in this paper.

	application runtime	specialization runtime
no runtime specialization	75.75s	0.00s
cold repository	63.19s	38.77s
repository with `energy` function	50.36s	23.43s
repository with `advance` function	35.00s	15.21s
repository with both functions	34.96s	0.00s

Fig. 10 Complete application runtimes (core 1) and compiler runtimes (core 2) without and with runtime specialization as well as for different initial states of the persistent specialization repository

8 Related Work

Our approach is related to a plethora of work in the area of just-in-time compilation; for a survey see [10]. Here, the general idea lies in the identification of hot spots in interpreted (byte) code during program execution. This is a common implementation strategy in many main-stream languages such as Java, C# or Python. Performance benefits mainly stem from running native machine code that avoids interpretive overhead and simplifying control structures. An extreme example is *tracing jit compilation* [11].

None of the major Java virtual machine implementations makes use of persistent compiled code repositories. The main arguments brought forward are that redoing the jit-compilation could even be faster than loading pre-generated code from disc and that disambiguating previously compiled code is a difficult semantic problem [12].

Sambamba [13] is an LLVM-based system that generates parallel executable binaries from sequential source code through runtime analysis of data dependencies. While this is conceptually similar to our system, the focus of Sambamba is on optimizing towards the runtime platform and not towards the data that is being worked with. Furthermore, the functional semantics of SaC statically solves many of the cases that Sambamba aims at with runtime compilation.

We shall also mention COBRA [14] (*Continuous Binary Re-Adaptation*). COBRA collects hardware usage information during application execution and adapts the running code to select appropriate prefetch hints related to coherent memory accesses as well as reduce prefetching to avoid system bus contention. The use of a controller thread managing optimization potential and a separate optimization thread applying the selected optimizations bears similarities with the organization of our adaptive specialization framework. One of the main differences between COBRA and our approach is that COBRA relies on information from hardware performance counters to trigger optimizations, whereas our approach triggers optimizations based on data format differences. Another difference is that COBRA, as the name suggests, works on binary executable code as input data, whereas we base our work on richly compiler-decorated intermediate code that gives us optimization opportunities on a much higher level of abstraction. Conversely, we are restricted to SaC as development platform, whereas COBRA works on any binary.

Another related project is Jikes RVM [15], an adaptive optimization system that monitors the execution of an application for methods that can likely improve application performance if further optimized. These candidates for optimization are put into a priority queue, which in turn is monitored by a controller thread. The controller dequeues the optimization request, forwards it to a recompilation thread which invokes the compiler and installs the resulting optimized method into the virtual machine. While this architecture matches our framework quite closely, the optimizations performed are entirely platform oriented, and not application or data oriented. Other similar systems include ADAPT [16], a system that uses a domain specific language to specify optimization hints that can be made use of at runtime, and ADORE [17], a predecessor of COBRA for single threaded applications.

More recently, Lutz and Grover [18] use just-in-time compilation in the context of C++ to dynamically optimize lambda functions used in STL algorithms. Very much like the SaC module system does, they store an internal representation of such functions in the compiled binary from where it is retrieved by the runtime system for optimization. Again in the context of C++, Haidl et al [19] use dynamic compilation to generate both host and device code for GPU computing from a single C++ source. Neither of these approaches appears to include persistent storage of binary clones though.

In contrast to all the above mentioned approaches, we adapt the intermediate code to properties of the (array) data it operates on, namely rank and shape. Our starting point is native binary code, not interpreted (byte) code. Last not least, the specific contribution of this paper, the persistence layer, appears to conquer mostly unchartered territory, and we have found rather little directly related work as far as persistence is concerned.

9 Conclusions and Future Work

Asynchronous adaptive specialization is a viable approach to reconcile the demand for generic program specifications in (functional) array programming with the need to achieve competitive runtime performance when compile time information about array ranks and shapes lacks. 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.

With our proposed persistence layer we demonstrate how asynchronous adaptive specialization overhead can drastically be reduced in practice. Following some training phase the vast majority of required specializations have already been generated in preceding runs of the same application or even independent applications with overlapping code base. If successful, pre-generated specializations merely need to be loaded from a specialization repository into a running application on demand. In practice the proposed persistence layer may effectively reduce the average overhead of asynchronous adaptive specialization to near nothing.

What appeared to be very attractive but mainly an engineering task at first glance, has proven to be fairly tricky from a conceptual perspective. 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. We sketched out our solutions found for each of these issues and thus have come up with a fairly complete account of the ins and outs of persistent asynchronous adaptive specialization for generic array programming in SaC.

It is noteworthy that while we explore our dynamic compilation approach in the context of SaC, our work is by no means entirely specific to SaC. In fact, much could be carried over to any context of data-parallel array processing. Interpreted array languages such as APL, J or MatLab are obvious candidates to look at, but we are not aware of any approaches similar to our's in these domains.

We pursue various directions of future work beyond the obvious: gaining more experience with our approach in practice. One such direction is the exploitation of platform-specific code generation opportunities that would adapt the running code not only to rank and shape of arrays, but at the same time to the exact processor, chip set and cache hierarchy of the machine. This was not the primary motivation of our work, but is likely to harness additional performance gains while technology-wise it would rather be a fairly simple by-product of our existing implementation. On the more technical side we work on controlling the size of our specialization repositories, which obviously cannot grow forever and for the time being require manual cleanup.

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
2. Grelck, C.: Single Assignment C (SAC): high productivity meets high performance. In Zsók, V., Horváth, Z., Plasmeijer, R., eds.: 4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary. Volume 7241 of Lecture Notes in Computer Science., Springer (2012) 207–278
3. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
4. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
5. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the gpu programming barrier with the auto-parallelising SAC compiler. In: 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA, ACM Press (2011) 15–24
6. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming. In Hammond, K., Loidl, H., eds.: Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK. Volume 7829 of Lecture Notes in Computer Science., Springer (2013) 279–294
7. Grelck, C., van Deurzen, T., Herhut, S., Scholz, S.B.: Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. Concurrency and Computation: Practice and Experience **24** (2012) 499–516
8. Grelck, C., Wiesinger, H.: Next generation asynchronous adaptive specialization for data-parallel functional array processing in sac. In: Implementation and Application of Functional Languages (IFL'13), Revised Selected Papers, ACM (2014)
9. Šinkarovs, A., Scholz, S., Bernecky, R., Douma, R., Grelck, C.: SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. Concurrency and Computation: Practice and Experience **26** (2014) 952–971 DOI: 10.1002/cpe.3078.
10. Aycock, J.: A brief history of just-in-time. ACM Computing Surveys **35** (2003) 97–113
11. Bolz, C., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the Meta-Level: PyPys Tracing JIT Compiler. In: 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09), Genova, Italy, ACM (2009) 18–25
12. JavaLobby: Why aren't JIT optimizations saved? http://www.javalobby.org/forums/thread.jspa?threadID=15812 (2004)
13. Streit, K., Hammacher, C., Zeller, A., Hack, S.: Sambamba: A runtime system for online adaptive parallelization. In: Compiler Construction (CC'12). Volume 7210 of LNCS., Springer (2012) 240–243
14. Kim, J., Hsu, W.C., Yew, P.C.: COBRA: An adaptive runtime binary optimization framework for multithreaded applications. In: International Conference on Parallel Processing (ICPP 2007). (2007)
15. Arnold, M., Fink, S., Grove, D., Hind, M., , Sweeney, P.F.: Adaptive optimization in the Jalapeño JVM. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00), Minneapolis, USA, ACM (2000)
16. Voss, M., Eigenmann, R.: High-level adaptive program optimization with ADAPT. In: ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, USA, ACM (2001) 93–102
17. Lu, J., Chen, H., Fu, R., Hsu, W.C., Othmer, B., Yew, P.C., Chen, D.Y.: The performance of runtime data cache prefetching in a dynamic optimization system. In: 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), San Diego, USA, IEEE (2003)
18. Lutz, T., Grover, V.: LambdaJIT: a dynamic compiler for heterogeneous optimizations of STL algorithms. In: 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC'14), Gothenburg, Sweden, ACM (2014) 99–108
19. Haidl, M., Steuwer, M., Humernbrum, T., Gorlatch, S.: Multi-stage programming for GPUs in C++ using PACXX. In: 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU'16), Barcelona, Spain, ACM (2016) 32–41