Next Generation Asynchronous Adaptive Specialization for Data-Parallel Functional Array Processing in SAC

Accelerating the Availability of Specialized High Performance Code

Clemens Grelck Heinrich Wiesinger

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

Abstract

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

In this paper we revisit asynchronous adaptive specialization in the context of the functional data-parallel array language SAC. We provide a comprehensive analysis of its strengths and weaknesses and propose improvements for its design and implementation. These improvements are primarily concerned with making specializations available to running applications as quickly as possible. We propose four complementary measures to this effect. Bulk adaptive specialization speculatively waits for future specialization requests to materialize instead of addressing each request individually. Prioritized adaptive specialization aims at selecting the most profitable specializations first. Parallel adaptive specialization reserves multiple cores for specialization and, thus, computes multiple specializations simultaneously. Last but not least, persistent adaptive specialization preserves specializations across independent program runs and even across unrelated applications.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IFL'13, August 28–30, 2013, Nijmegen, The Netherlands.
Copyright © 2013 ACM 978-1-4503-2988-0/13/10…$15.00.
http://dx.doi.org/10.1145/2620678.2620690Reprinted from IFL'13, Implementation and Application of Functional Languages, August 28–30, 2013, Nijmegen, The Netherlands, pp. 117–128.

1. Introduction

SAC (Single Assignment C) is a purely functional, data-parallel array programming language [9, 13, 15]. As such, SAC puts the emphasis on homogeneous, multi-dimensional arrays as the most relevant data aggregation principle. SAC advocates shape- and rank-generic programming on multi-dimensional arrays, i.e. SAC supports functions that abstract from the concrete shapes and even from the concrete ranks (number of dimensions) of argument and result arrays. Depending on the amount of compile time *structural* information we distinguish between different runtime representations of arrays.

From a software engineering point of view it is (almost) always desirable to specify functions on the most general input type(s) to maximize code reuse. For example, a simple structural operation like rotation should be written in a rank-generic way, a naturally rank-specific function like an image filter in a shape-generic way (i.e. for 2-dimensional arrays). Very infrequently it can be desirable to write code in a non-generic way. Consequently, the extensive SAC standard library is full of generic, mostly rank-generic functions.

However, genericity comes at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again for rank-generic code [25]. 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 [12, 14] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Third, in automatically parallelized code [5, 7, 8, 18] 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 [16, 17]. The idea is to postpone specialization, if necessary, until runtime, when complete structural information is always 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 function and dynamically links the additional code into the running application program. Eligible functions are indirectly dispatched such that if the same binary function is called again with arguments of the same shapes as previously, the corresponding new and fast non-generic clone is run instead of the old and slow generic one.

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

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. The contribution of this paper, in addition to a comprehensive analysis of the situation, is to investigate optimizations and extensions of our framework proposed in [16, 17] to this effect. These fully complementary extensions fall into four categories:

- Our first approach is to combine multiple specialization requests to be served in one compiler run. Here we aim at reducing dynamic compilation times per specialized function by harnessing common synergy effects in compilation.
- Our second approach is to prioritize those specialization requests that promise the highest return on investment, where the investment is dynamic compilation time and the return is the relative improvement of execution time comparing specialized and generic versions of the same function.
- Our third approach is to parallelize the specialization controller in order to produce multiple specializations concurrently. Instead of a fixed classification of the host architecture's cores as either compute cores or specialization cores, we propose a demand-driven dynamic adjustment of hardware resources.
- Our fourth approach is to make specializations persistent across multiple runs of the same application or even across multiple unrelated applications that make use of an overlapping set of libraries.

All four approaches are orthogonal and can be combined without restriction. Jointly, they define a comprehensive and ambitious research agenda in the area of dynamic compilation.

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 runtime specialization framework in more detail. Sections 4, 5 and 6 illustrate the benefits of our approach in general by means of three case studies: generic convolution, matrix multiplication and n-body simulation, respectively. They are followed by a comprehensive analysis of the strengths and weaknesses of asynchronous adaptive specialization in Section 7. In Sections 8, 9, 10 and 11 we discuss the above four research directions. Finally, we sketch out some related work in Section 12 and draw conclusions in Section 13.

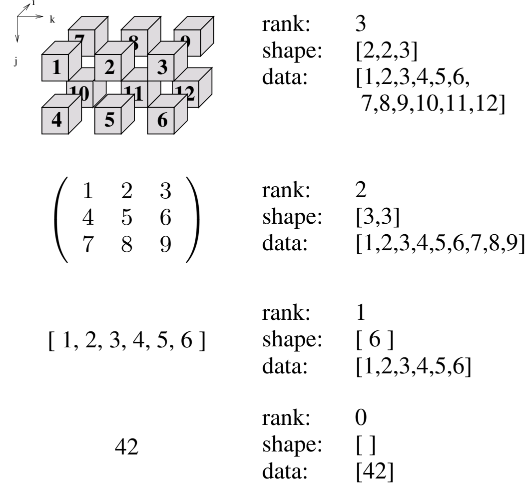

Figure 1. Truly multidimensional arrays in SAC and their representation by data vector, shape vector and rank scalar

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 [9, 13].

Following the example of interpreted array languages, such as APL[6, 20], J[21] and NIAL[22, 23], 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.

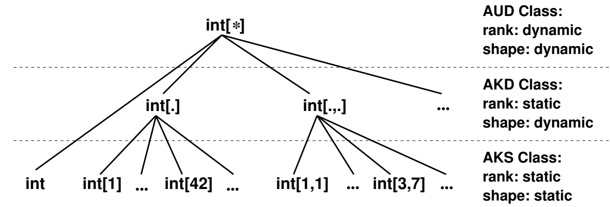

Figure 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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

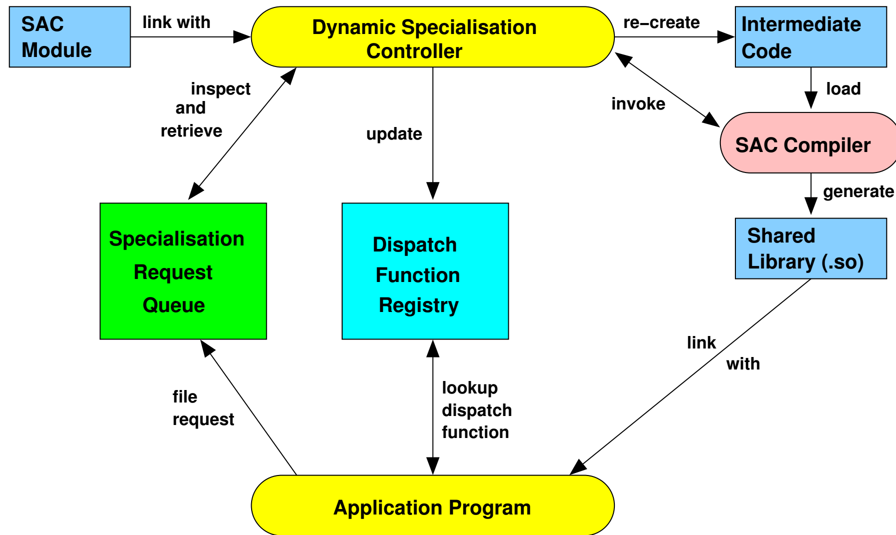

Figure 3. Software architecture of asynchronous adaptive specialization framework

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

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

In the sequel we demonstrate potential runtime behaviour of applications in the context of dynamic specialization by three case studies. We begin with a generic convolution kernel with convergence check in the next section and continue with matrix multiplication and n-body simulation in Sections 5 and 6, respectively.

4. Case study 1: generic convolution

Fig. 4 shows a SAC module `ConvolutionAuxiliaries` that defines and exports two rank-generic functions: `convolution_step` and `is_convergent`. The former defines a single convolution step that computes each element of a multi-dimensional grid as the arithmetic mean of its direct neighbours along each axis. The latter implements a predicate whether or not all corresponding elements of two given arrays differ by less than a given threshold. Due to using the `rotate` function imported from the comprehensive SAC array library this convolution step implements cyclic boundary conditions. The C-style `for`-loop is merely syntactic sugar for an inlined tail-recursive anonymous function, and `tod` refers to a conversion function from integer numbers (`int`) to double precision floating point numbers (`double`).

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

Figure 4. Case study: SAC module that exports generic convolution step and convergence check functions

Fig. 5 shows a second module named `Convolution`. This module defines and exports a single function named `convolution`, which computes a series of convolution steps until sufficient convergence is reached. More precisely, in every iteration of the `do/while`-loop (again syntactic sugar for an inlined tail-recursive anonymous function) the function `convolution` applies both imported functions `convolution_step` and `is_convergent`. We interpret C-style statement sequences as nested `let`-expressions, and consequently support repeated assignment to the apparently same variable. However, these are in fact different purely functional place-holder variables that bear the same name.

A more detailed description of the compositional style of array programming advocated by SAC along with a more thorough explanation of a variant of the code shown here can be found in [9].

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

Figure 5. Case study: SAC module that implements a generic convolution kernel with convergence check based on the functions shown in Fig. 4

Since this is not a paper about programming in SAC or the language design of SAC, we refrain from repeating this information here and refer the interested reader to the above resource.

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

A representative plot of the runtimes achieved is shown in Fig. 6. It reports on a convolution experiment with a 3-dimensional array of $100 \times 100 \times 100$ double precision floating point numbers. The figure shows individual iterations on the x-axis and measured execution time for each iteration on the y-axis. The two lines show measurements with runtime specialization disabled and enabled, respectively. We run the application code sequentially on one core and the asynchronous adaptive specialization controller on a second core. Thus, what the graphs in Fig. 6 do not show is competition of application threads and specialization controller for limited computing resources. We leave such experiments to future work.

Instead, we can read from the graphs that for the given example runtime specialization does not inflict any measurable overhead in the startup phase and while the specialization controller is still working on the first specialization.

After 8 iterations running completely generic binary code a shape-specialized version of the `convolution_step` function becomes available. Switching from a generic to a non-generic implementation of the convolution step reduces the execution time per iteration from about 1.5 seconds to roughly 0.25 seconds. This example demonstrates the tremendous effect that runtime specialization can have on generic array code.

The 3-dimensional case requires a total of six rotations of the argument array. Rotation is not a built-in function in SAC, but itself is implemented using two consecutive basic array operations (with-loops). Rank-generic binary code cannot further be optimized and leads to a total of 19 intermediate arrays to compute the final result of a single convolution step. For the specialized intermediate

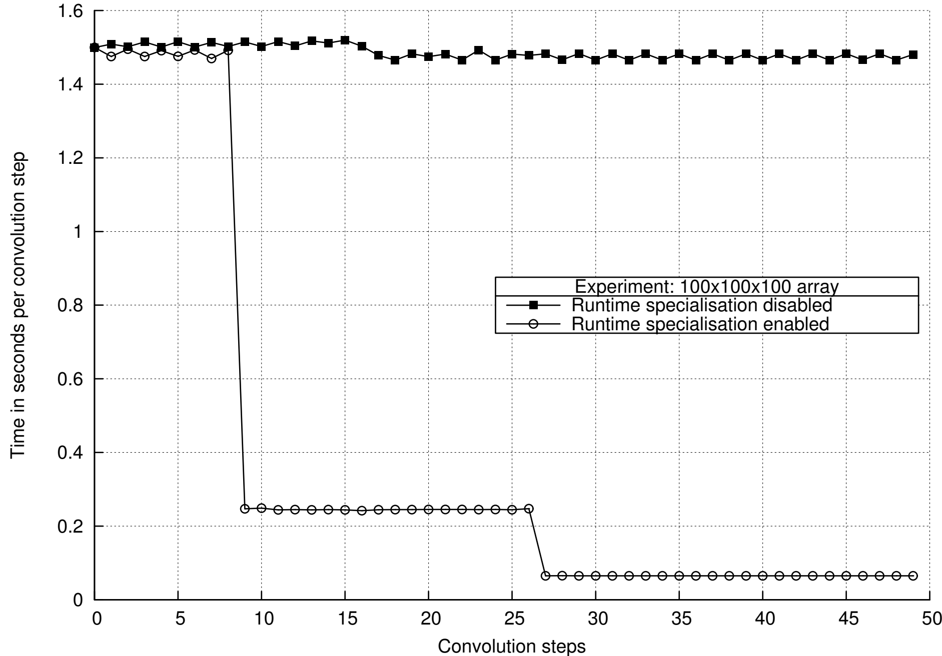

Figure 6. Case study: running the generic convolution kernel defined in Fig. 4 and Fig. 5 on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialization

code the compiler unrolls the `for`-loop three times due to the three dimensions of the chosen example. As a consequence of this instance of loop unrolling the rotation axes, previously given by the induction variable `i`, become constants. This enables a series of partial evaluations in the inlined definitions of the `rotate` function and ultimately the fusion of all `with`-loops in the intermediate representation of the convolution step. Consequently, the entire convolution step will be computed in one step, and not a single intermediate array materializes in memory.

As soon as the specialization of the convolution step is completed, the specialization controller starts working on the already pending specialization request for the convergence check. As illustrated in Fig. 6, the specialized binary code for the convergence check becomes available after 26 iterations and reduces the execution time of a single iteration further from 0.25 seconds to 0.065 seconds. The main reason for this considerable performance improvement again is the effectiveness of optimizations that fuse consecutive array operations and, thus, avoid the creation of intermediate arrays. In the relatively simple case of the convergence criterion, as shown in Fig. 4, we could in principle fuse the whole pipeline of basic array operations without knowing the concrete rank and shape because they are effectively simple homogeneous zip-, map- and reduce-operations over the entire index space of the original argument arrays. However, our original version of *with-loop folding* [27] does not cover rank- or shape-generic code and more recent extensions [4] are not yet fully operational.

With all binary code specialized for the relevant array shape $100 \times 100 \times 100$ no further improvements are to be expected for the remainder of the application runtime. This represents a very desirable case for our approach, meaning we reach a fixed point in dynamic code adaptation. Consequently, the remainder of program execution benefits from the adapted code without continuously causing further overhead for on-going runtime specialization.

5. Case study 2: matrix multiplication

In order to demonstrate that the benefits of asynchronous adaptive specialization are not specific to the convolution kernel illustrated in the previous section we now look at a different case study: matrix multiplication. Fig. 7 shows a possible SAC definition of matrix multiplication that makes heavy use of the so-called *axis control notation*. For any details about the latter we refer the interested reader to [11] for a comprehensive coverage. Given the simplicity of the code and assuming general familiarity with matrix multiplication as numerical kernel, we believe the code can be understood straightforwardly. First, we transpose the second argument matrix. Then, we compute for each element of a 2-dimensional index space bounded by the number of rows of argument matrix `A` and the number of columns of argument matrix `B` (or the number of rows of its transpose `Bt`) the sum of element-wise product of the corresponding rows of `A` and `Bt`.

```
1   module MatMul;
2
3   use Array: all;
4
5   export {matmul}
6
7   double[.,.]
8   matmul (double[.,.] A, double[.,.] B)
9   {
10    Bt = {[i,j] -> mat[j,i]};
11
12    return {[i,j] -> sum( A[i] * Bt[j])};
13  }
```

Figure 7. Case study: SAC module that exports a matrix multiplication function

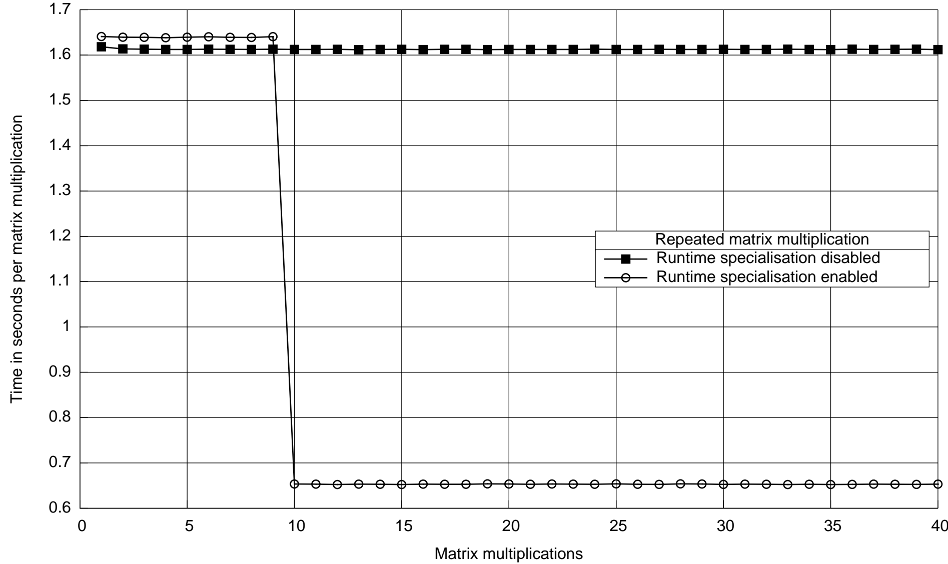

Figure 8. Case study: repeatedly running the matrix multiplication kernel defined in Fig. 7 and Fig. 9 on matrices of shape 1000×1000 with and without asynchronous adaptive specialization

To demonstrate any positive effect of asynchronous adaptive specialization we must repeatedly multiply matrices of the same shape. Fig. 9 shows a suitable benchmark code. In a real-world scenario we would expect matrix products to be more interleaved with other computations. This would, of course, reduce the visible effect of our efforts, depending on the ratio of operations in the code, but not change the situation in principle.

```
1  double[.,.]
2  multi_matmul( double[.,.] A,
3                double[.,.] B,
4                int iter)
5  {
6    for (i=0; i<iter; i++) {
7      A = matmul( A, B);
8    }
9
10   return A;
11 }
```

Figure 9. Case study: SAC benchmark code that implements repeated matrix multiplication based on the definition shown in Fig. 7

This time we run our experiments on a large 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 processor core has 64 KB of L1 cache for instructions, 64 KB of L1 cache for data, and 512 KB of L2 cache. Each group of 6 cores shares one L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5.

Fig. 8 shows the outcome of our experiments for matrices of shape 1000×1000 in a similar style as Fig. 6 for the generic convolution kernel. In fact, despite the numerical differences we observe almost the same dynamic behaviour for both applications. The overhead of asynchronous adaptive specialization in the beginning is reproducible but marginal. For the given problem size, we link with the specialized variant of the matrix multiplication function after nine iterations of the benchmark code and reduce the (fairly stable) execution time from 1.6 seconds per benchmark iteration to about 0.65 seconds.

The reason for this improvement lies in the successful fusion of the inner kernel that defines the value of each element of the product matrix. Here, we select one row of matrix A and one row of the transposed matrix B before we compute the element-wise product. This leads to the creation of three temporary vectors prior to the final reduction operation (sum). In the specialized version of the matmul function we are again able to fuse these various operations and to avoid effective creation of temporary arrays at runtime.

6. Case study 3: n-body simulation

Our third case study, n-body simulation, has extensively been studied and discussed in [31]. Therefore, we restrict ourselves here to present the top-level driver function, which can be found in Fig. 10. The main compute effort is hidden within the function advance that computes one step of the n-body simulation. More precisely, it computes a new vector of body positions and a new vector of body velocities based on old positions and old velocities as well as the individual masses of the bodies and a given threshold. Both positions and velocities are defined as triples of floating point numbers corresponding to the three spatial dimensions.

To make the example more interesting for our purpose we compute the collective energy of the simulated system before and after the simulation itself by means of the energy function and print the results to the standard output. Regarding purely functional input/output with a C-like look-and-feel we refer the interested reader to [9, 10] for more information.

In Fig. 11 we show experimental results for our n-body simulation obtained on the same AMD Opteron Magny-Cours system as used in the previous section for matrix multiplication. We study two problem sizes: 768 bodies and 1024 bodies. We can observe a very similar overall behaviour as in the previous experiments on generic convolution in Section 4 and on matrix multiplication in Section 5. As expected, the execution time per step is higher for 1024 bodies than for 768 bodies, and thus it takes less steps for the asynchronously specialized, better performing variant to become

```
1  double[.,.], double[.,.]
2  nbody_sim( double[.,.] body_pos, double[.,.] body_vel, double[.] body_mass,
3             int steps, double threshold)
4  {
5    printf ("%.9f\n", energy( body_pos, body_vel, body_mass));
6
7    for (i=0; i<steps; i++) {
8      body_pos, body_vel = advance( body_pos, body_vel, body_mass, threshold);
9    }
10
11   printf ("%.9f\n", energy(  body_pos, body_vel, body_mass));
12
13   return (body_pos, body_vel);
14 }
```

Figure 10. Case study: excerpt from a SAC implementation of n-body simulation; for more details see [31]

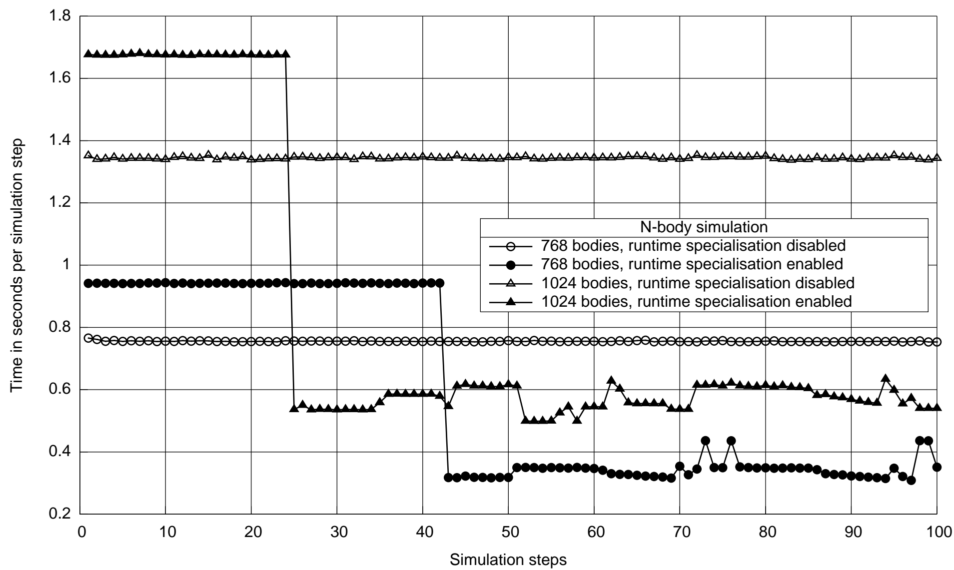

Figure 11. Case study: running the n-body simulation kernel illustrated in Fig. 10 and discussed intensively in [31] with and without asynchronous adaptive specialization

available. Of course, the time it takes to compile the specialization is independent from the problem size. We attribute the variation in per-step execution times after specialization as measurement artifacts and consider them irrelevant in our context.

7. Analysis

As pointed out earlier, the time it takes to make an asynchronously specialized binary variant of some originally generic function available to a running application is constant (for a given intermediate code, compiler version and compiler options). In contrast, the time it takes to complete one iteration of an application depends on the rank and shape of the argument array(s).

Running the very same application on a considerably larger problem size may lead to a situation in which all specializations become available long before the application moves on to the second iteration. The other extreme is likewise possible: For small problem sizes the entire application may have terminated before even the first specialization becomes available. In this case, the specialization controller discards the specialization attempt and terminates with the application itself.

As the three case studies demonstrate, the relative performance difference between generic and specialized binary variants of the same function can be very relevant. How relevant it is in practice, crucially depends on the individual programming style. For the generic, compositional programming style advocated by SAC (see [9]) and applied in the three case studies we can generally expect a high performance gain due to specialization, may it be static where possible or dynamic where needed. It must, however, be said as well that where functions are defined in a more low-level, direct and application-specific style, the gain could likewise be marginal.

As a matter of principle, asynchronous adaptive specialization is most effective for long-running applications. Therefore, any measure that contributes to making specialized binary variants available to a running application more quickly is beneficial in practice and improves the applicability of the entire approach.

A number of aspects affect the time that it takes from filing a specialization request by the running application to the specialized binary effectively becoming available for dispatch. The most relevant aspect in one way or another is the execution speed of the compiler. For good reasons the design of the SAC compiler is di-

ametrically opposed to that of typical just-in-time compilers for byte-code interpreted languages. Whereas the latter are optimized for short compilation times, the SAC compiler has from the very beginning been optimized for speed of compiled code, not speed of the compiler.

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

In the sequel we investigate four approaches to effectively reduce the time to availability of dynamically specialized functions that can be pursued without touching the SAC compiler as a whole.

8. Bulk Asynchronous Adaptive Specialization

Our first research direction focuses on the way the specialization controller retrieves specialization requests from the specialization request queue. Our initial approach, as detailed in Section 3, follows the straightforward approach to service one request after the other. In practice, however, it may be considered fairly common that multiple specialization requests residing in the queue actually refer to functions from the same module or even to the same function to be specialized for different argument shapes. In either case we can potentially harness substantial synergy effects by combining several individual specialization requests into one combined request prior to servicing it.

These synergy effects stem from a variety of sources. First of all, specializing the same function twice still only requires the re-instantiation of the precompiled intermediate code representation once. More importantly, a SAC module typically imports a considerable amount of external symbols, may they come from other application specific modules or the extensive SAC standard library. Many of these symbols are inline functions whose precompiled intermediate code is incrementally made available to the dynamic compilation process, very much with the same techniques that allow us to retrieve intermediate representations of our specialization candidates.

In practice, compilation of a SAC module typically attracts considerable amounts of intermediate code beyond the actual specialization candidates. All this additional code, however, is exactly the same no matter how many times we specialize the same function for different argument shapes. Even if we aim at specializing different functions from the same module, the described effect is more than noticeable. In typical cases a large fraction of the imported intermediate code can be shared between specialization candidates. In practice, we can indeed observe that the number of specializations (within reason) has a minor effect on compilation times. Thus, combining multiple specializations in one compilation process appears to be an attractive option.

There are, however, a few complications to be observed. Looking at our first case study discussed in Section 4, the generic convolution kernel with convergence check, we must admit that at least a straightforward realization of the proposed approach does not yield the desired advantages. The reason for this is simple: each time the specialization controller retrieves a request from the request queue the queue has exactly one entry. The first time the example application files a specialization request the specialization controller is eagerly waiting for it and immediately retrieves it. The second time the application files a specialization request this is already the final one before the application reaches a fixed point with respect to dynamic adaptation.

Of course, we can expect from less simple applications that when the specialization controller completes the first specialization, it will indeed find a request queue with multiple entries. Notwithstanding, our case studies highlight an issue: the greedily waiting specialization controller. In many scenarios it may actually prove advantageous to wait a certain (relatively short) period of time to give further specialization requests a fair chance to effectively materialize in the request queue.

Obviously, postponing the first dynamic specialization has a speculative character. If the whole application merely creates a single specialization request during its entire runtime then we waste the additional waiting time, and this approach has a small and bounded detrimental effect on application performance. Of course, in many cases some static code analysis could clarify the situation.

9. Prioritized Asynchronous Adaptive Specialization

Our second approach, like the first, is concerned with how the specialization controller retrieves requests from the request queue. Whereas our previous approach is concerned with the combination of multiple requests involving the same specialization candidate or, at least, the same module of origin, we now look into unrelated specialization requests to functions from different modules. This scenario does not allow us to exploit similar synergy effects as in the previous case, but it gives us an opportunity to consciously select one specialization candidate rather than simply taking the first from the queue. There is no semantic requirement to process requests in the order that they originate during the execution of an application. Thus, the specialization controller could likewise traverse the entire queue, sort it into buckets referring to the same module and then make a choice which bucket appears to be the most promising in terms of expected return on investment.

Which bucket is the most promising is a-priori undecidable. However, we expect simple heuristics to already have a measurable positive effect. For instance, we can choose the bucket with the largest number of functions. On the more sophisticated side we can monitor both the dynamic compilation time of functions as well as their execution time prior to and after specialization. On the basis of such monitoring information we can at least make a a fairly well educated guess. It goes without saying that, as in the first approach, any attempt to predict the performance of still to be specialized functions inevitably involves an element of speculation.

Another basis for prioritization is the (predicted) number of potential applications of some dynamically specialized function. Our third case study, n-body simulation as described in Section 6, gives a good example for this. In our initial approach, as explained in Section 3, the first dynamic specialization request is for function `energy`, but this will only be applied once more at the end of the simulation. At the same time, specialization of the much more relevant function `advance` that is repeatedly used in the loop is delayed. This could be identified by static code analysis and dynamic specialization of the `advance` function consequently be prioritized over dynamic specialization of the `energy` function.

10. Parallel Asynchronous Adaptive Specialization

With compute cores promised to be available in abundance in the near future, if not already today, the same argument that we used to motivate setting aside one core for specialization instead of data-parallel execution of the application likewise holds for more than one core. Looking back at Fig. 6, Fig. 8 and Fig. 11 we observe that

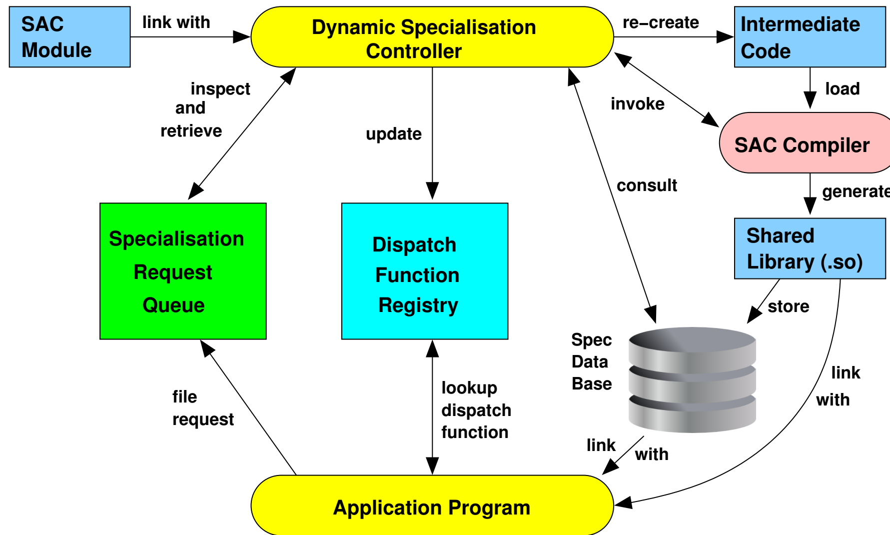

Figure 12. Software architecture of asynchronous adaptive specialization framework with persistent storage

the relative performance improvements realized by adaptive asynchronous specialization by far outweigh potential improvements through data-parallel execution even under the assumption of linear speedups. On systems with tens of cores parallel specialization through multiple concurrent specialization controllers should be beneficial even if the relative performance improvements achieved are less impressive.

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

In other words, it proves to be rather unfortunate that we first specialize the convolution step and only after completing this task turn towards the convergence check. If we would specialize the convergence check first, partially specialized code would already be available after 3–4 iterations. Unfortunately, the specialization order is beyond our control because the generic implementation of the convolution step is simply run before that of the convergence check in the application code (unless we would apply prioritized specialization, as described in the previous section).

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

While parallelizing asynchronous adaptive specialization appears to be beneficial no matter what if only sufficiently many compute cores are available, the question arises how many cores would be best to use for specialization and how many for data-parallel execution of the application program itself. For our first case study, looked at in more detail here, this question seems to be straightforward to answer: two. However, even for this admittedly simple application this is not the optimal number. Once both specializations have been created, the two specialization controllers would wait in vain for any other requests to come and thus would waste two compute cores until the termination of the application. It seems plausible that these two cores should rather help running the application, in particular on a small quad-core system as we used for these experimentats but also on a larger system as we had access to for the further experiments.

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

11. Persistent Asynchronous Adaptive Specialization

Our last area of refinement lies in making asynchronous adaptive specializations persistent. So far specializations are accumulated during the execution of an application, but 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.

Therefore, we propose to store specialized binary functions in persistent collections alongside the original generic binary modules. Fig. 12 shows a sketch of the extended framework architecture. The most notable difference to Fig. 3 is the database of specialized functions shown in the lower right corner. Whenever a new specialized binary version of some generic function is created, it is not only linked into the running application that requested this particular specialization, but it is additionally stored in the database.

The other area that needs refinement is the specialization controller. Instead of checking only for potentially existing specializations created previously in the same application run or currently pending, the specialization controller additionally consults the external persistent database to figure out whether or not the required specialization already exists. Depending on the outcome of this

query the application either dispatches to the specialized implementation immediatetely or files a specialization request to be taken care of by a specialization controller.

The main advantage of persistent storage is that the overhead of actually compiling specializations at application runtime can often be avoided. Our assumption is that for many applications the proposed approach in practice results in a sort of training phase, after which most required specializations have become available. 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.

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

12. Related Work

Our approach is related to a plethora of work in the area of just-in-time compilation; for a survey see [3]. Our work, however, differs from just-in-time compilation of (Java-like) byte code in several aspects. In the latter hot spots of byte code are adapted to the platform they run on by generating native code at runtime, whereas the execution platform was deliberately left open at compile time. This form of adaptation (conceptually) happens in a single step [3]. In contrast, our approach adapts code not to its execution environment but to the data it operates on. This is an incremental process that may or may not reach a fixed point. The number of different array shapes that a generic operation could be confronted with is in principle unbounded, but in practice the number of different array shapes occurring in a concrete application is often fairly limited.

There are of course also projects that use just-in-time compilation for iterative runtime optimizations. One such project is Sambamba [28], an LLVM based system that generates parallel executable binaries from sequential source code. Such optimization can only be partially applied at compile time because of data dependencies that are only fully known at runtime. Sambamba generates optimization hints at compile time that can be used by a runtime component to further optimize the code based on the then available information. While this is conceptually similar to our system, the focus of Sambamba is still on optimizing towards the runtime platform and not towards the data that is being worked with.

A step closer to our proposed system is COBRA [24] (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 the aggressiveness of prefetching to avoid system bus contention. The concept here, while still focusing on optimization towards the platform the code runs on, is already also taking the data that is being worked on into account. The architecture employed for performing those optimizations shows similarities to our approach as well. Specifically the use of a controller thread managing optimization potential and a separate optimization thread applying the selected optimization. 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 on data format differences.

Another project with architecture similarities to our proposed system is Jikes RVM [1, 2]. Jikes RVM has 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 in 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 system already quite closely, the optimizations performed are still platform oriented.

Other similar systems include ADAPT [29, 30], a system that uses a domain specific language to specify optimization hints that can be made use of at runtime, ADORE [26], a predecessor of COBRA for single threaded applications, and earlier work on an adaptive runtime optimization system for FORTRAN [19].

It is noteworthy that while we explore our dynamic compilation approach in the context of the functional data-parallel language SAC, our work is not specific to SAC, but can be carried over to any context of data-parallel array processing.

13. 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 this paper we have proposed several improvements and extensions to asynchronous adaptive specialization that generally broaden its applicability by making specialized binary code available quicker. Persistent asynchronous adaptive specialization aims at sharing runtime overhead across several runs of the same application or even across multiple independent applications sharing some library code and thus to effectively eliminate the observable overhead in many situations (Section 11). The parallelization of the specialization process itself with a variable distribution of cores between specialization and data-parallel application execution allows us to satisfy specialization requests as quickly as possible (Section 10). Combining multiple specialization requests in one dynamic compilation process helps to reduce the effective overhead per specialization (Section 8), and consciously selecting the most promising specialization requests (prioritized asynchronous adaptive specialization, Section 9) aims at making the best possible use of the available computing resources.

We are occasionally asked whether it would not be sufficient to statically compile prophylactic specializations of generic functions to arguments with, say, one, two and three dimensions and keep their shape dynamic. This, however, is not the case for essentially two reasons. Firstly, functions are not limited to a single parameter. Hence, we would need to speculatively specialize each function to all combinations of potential argument ranks. Already for a function with three arguments specialization to ranks one, two and three would result in 27 different specialized versions. More parameters and more dimensions quickly lead to an undesirable combinatoric explosion of variants.

Secondly, non-trivial intermediate code representations, e.g. the one stemming from the convolution step example with the multiple rotations (see Section 4), do in practice very much benefit from the availability of concrete shapes for the effectiveness of our various compiler optimizations [12, 14]. Without the ability to continuously simplify intermediate code through partial evaluation, the various index subdomains resulting from repeated intersection of existing index subdomains quickly become intractable.

In this paper we demonstrate the effects of runtime code adaptation by means of three case studies that, despite implementing very different numerical algorithms, expose very similar runtime behaviour under asynchronous adaptive specialization. This demonstrates the wide applicability of the proposed techniques.

Indeed, their benefit stems from the runtime performance discrepancy of generic and specialized binary code alternatives. This discrepancy, in turn, very much depends on programming style. With the compositional programming methodology advocated by SAC and illustrated in the three case studies the difference is typically substantial. The whole optimization potential of the SAC compiler is required to systematically transform code from a form that is amenable to software engineering into a form that allows for efficient execution on computing machinery. The practical effectiveness of many of these intermediate code optimizations critically depends on the availability of rank and shape information. In this case, performance improvements of more than an order of magnitude, as demonstrated by the case studies, is not only possible but can be considered typical.

With a more low-level programming style in SAC, where concrete algorithms are implemented in a more direct style using with-loops (SAC's underlying versatile array comprehension and reduction construct), the performance difference could be much smaller. In this case, performance improvements would entirely stem from different runtime representations of arrays and could be not more than about 10–15%.

The main insight here is that the benefit of asynchronous adaptive specialization and the proposed refinements does only marginally depend on the concrete problem solved by some (fragment of) SAC code. Instead, it critically depends on the individual programming style used in an application. Following the compositional style of programming advocated by SAC and used throughout the three case studies would, under a naive compilation regime, lead to large numbers of temporary arrays. This gives rise to large-scale compiler optimization and, consequently, a substantial difference between original and optimized code. The effectiveness of optimization, however, critically depends on the availability of rank and shape information. Therefore, we can expect a substantial performance difference. In contrast, with code written in a low-level, direct style the performance gain of asynchronous adaptive specialization might end up being only marginal.

A positive side-effect of asynchronous adaptive specialization is that we can rather on-the-fly also adapt dynamically generated code to the concrete compute architecture the application is running on. In many dynamic compilation settings this sort of adaptation is the primary motivation. In our case, however, it is rather a side product. Dynamic generation of code customized to a specific architecture in our case would merely require the invocation of the back-end C compiler with specific parameters, while the mostly target architecture agnostic compilation from SAC to C would mostly remain unaffected. In the experiments described in Sections 4 through 6 we have not yet made use of this opportunity. So, additional performance gains could still be realized wth negligible effort on our compiler and specialization framework.

We are currently busy working on implementing the various proposed improvements for asynchronous adaptive specialization. Our future work, hence, is dominated by completing this implementation and conducting extensive experiments to evaluate the benefits of the proposed extensions in detail.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, , and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00), Minneapolis, USA*. ACM, 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, , and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005.

[3] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[4] R. Bernecky, S. Herhut, and S.-B. Scholz. Symbiotic Expressions. In M. T. Morazan and S.-B. Scholz, editors, *Implementation and Application of Functional Languages, 21st International Symposium, IFL 2009, South Orange, NJ, USA*, number 6041 in Lecture Notes in Computer Science, pages 107–126. Springer, 2011.

[5] M. Diogo and C. Grelck. Towards heterogeneous 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*, pages 279–294. Springer, 2013.

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

[7] 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.

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

[9] 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.

[10] C. Grelck and S.-B. Scholz. Classes and Objects as Basis for I/O in SAC. In T. Johnsson, editor, *7th International Workshop on Implementation of Functional Languages (IFL'95), Båstad, Sweden*, pages 30–44. Chalmers University of Technology, Gothenburg, Sweden, 1995.

[11] C. Grelck and S.-B. Scholz. Axis Control in SAC. In R. Peña and T. Arts, editors, *Implementation of Functional Languages, 14th International Workshop (IFL'02), Madrid, Spain, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2003.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] G. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. PhD thesis, Carnegie-Mellon University, Pittsburgh, USA, 1974.

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

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

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

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

[24] J. Kim, W.-C. Hsu, and P.-C. Yew. Cobra: An adaptive runtime binary optimization framework for multithreaded applications. In *International Conference on Parallel Processing (ICPP 2007)*, 2007.

[25] 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.

[26] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. 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.

[27] S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer, 1998.

[28] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In M. O'Boyle, editor, *21st International Conference on Compiler Construction (CC'12), Tallinn, Estonia*, volume 7210 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 2012.

[29] M. Voss and R. Eigenmann. A framework for remote dynamic program optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00), Boston, USA*, pages 32–40. ACM, 2000.

[30] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, USA*, pages 93–102. ACM, 2001.

[31] A. Šinkarovs, S. Scholz, R. Bernecky, R. Douma, and C. Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience*, 26(4):952–971, 2014. DOI: 10.1002/cpe.3078.