Advances in Dynamic Compilation for Functional Data Parallel Array Processing

CLEMENS GRELCK, University of Amsterdam
HEINRICH WIESINGER, University of Amsterdam

Data-parallel processing of multi-dimensional 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 contradicting demands.

In this paper we report on our on-going work on dynamic compilation in the area of data-parallel array processing in the context of the purely functional array language SAC. This work is currently mainly concerned with making dynamically specialized function alternatives available to the running application as quickly as possible in order maximize the positive performance impact of dynamic compilation. To this effect We propose (and work on) a bundle of measures: prioritization of asynchronous adaptive specialization, combined specialization of multiple candidates, parallel adaptation taking multiple cores away from the application and, last not least, making specializations persistent.

ACM Reference Format:
Clemens Grelck, Heinrich Wiesinger, 2014. Advances in Dynamic Compilation in Functional Data Parallel Array Processing. *ACM* V, N, Article A (January 2014), 12 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

SAC (Single Assignment C) is a purely functional, data-parallel array programming language [Grelck and Scholz 2006b; Grelck 2012] that 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. Very infrequently it is desirable to write code in a non-generic way. Consequently, the extensive SAC standard library is full of generic, mostly rank-generic functions.

Author's addresses: C. Grelck, H. Wiesinger, Universiteit van Amsterdam, Instituut voor Informatica, Science Park 904, 1098XH Amsterdam, Netherlands

DOI: http://dx.doi.org/10.1145/0000000.0000000

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 [Kreye 2002]. The reasons are manifold and often operation-specific, but three categories can be identified nonetheless: Firstly, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Secondly, many of the SAC compiler's advanced optimizations [Grelck and Scholz 2003; 2006a] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Thirdly, in automatically parallelized code [Grelck 2005; Guo et al. 2011; Diogo and Grelck 2013] 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 [Grelck et al. 2010; 2012]. The idea is to postpone specialization, if necessary, until runtime, when full structural information is always available. Asynchronous with the execution of a generic function, potentially in a data-parallel fashion on multiple cores, a *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. Taking adaptive specialization out of an application's critical path opens up an avenue to apply such techniques in conjunction with highly optimizing but slow compilers.

The effectiveness of our approach, nonetheless, depends on making specialized, and thus considerably more efficient, binary variants available to a running application as quickly as possible. We do primarily target longer running applications, but at the same time our compiler is also a highly-optimizing one with considerable compilation times, not a lean just-in-time compiler designed for dynamic compilation. The SAC compiler contains more than 200 passes, both lowering steps and optimizations, where the latter are arranged in cycles to compute fixed points on the intermediate code representation (an overview on the compiler structure can be found in [Grelck 2012]). Of course, compilation times could be drastically reduced by switching off some or all of the optimizations, but the difference in runtime performance exactly from this optimization capacity, and, hence, switching them off would be counter-productive. Thus, time to availability of dynamic specializations is a crucial issue. The contribution of this paper is to investigate optimizations and extensions of our framework proposed

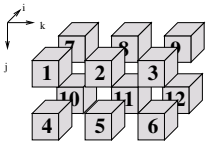
in [Grelck et al. 2010; 2012] to this effect. We look into four complementary but interrelated approaches:

(1) Manifold adaptive specialization speculatively waits for future specialization requests to materialize instead of addressing each individually.
(2) Prioritized adaptive specialization aims at selecting the most profitable specializations first.
(3) Parallel adaptive specialization reserves multiple cores for specialization instead of application processing and computes multiple specializations simultaneously.
(4) Persistent adaptive specialization preserves specializations across independent program runs and even across unrelated applications.

The remainder of the paper is organized as follows. We first provide some more background on functional data parallel array processing with SAC in general (Section 2) and on asynchronous adaptive specialization in particular (Section 3). We elaborate further on the four proposed optimizations (Sections 4, 5, 6 7) before we sketch out related work (Section 8) and conclude (Section 9).

2. SAC AND THE CALCULUS OF MULTI-DIMENSIONAL ARRAYS

As the name "Single Assignment C" suggests, SAC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This choice is primarily meant to facilitate 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 [Grelck and Scholz 2006b; Grelck 2012].



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

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

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

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

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

42

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

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

Following the example of interpreted array languages, such as APL[Falkoff and Iverson 1973; International Standards Organization 1993], J[Iverson 1991] or

NIAL[Jenkins 1989; Jenkins and Glasgow 1989], 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.

Fig. 1 illustrates the calculus of multi-dimensional arrays that is the foundation of array programming in SAC. The array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Consequently, every value in SAC has rank, shape vector and data vector. Both rank and shape vector can be queried by built-in functions of the same name. The data vector can only be accessed element-wise through a selection facility adopting the square bracket notation familiar from C-like languages. Given the ability to define rank-generic functions, whose argument arrays' 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.

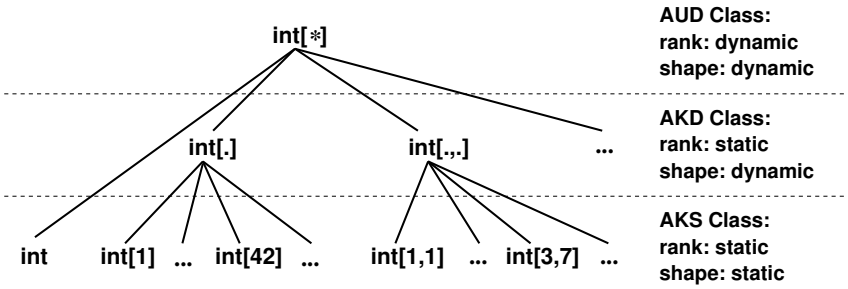


Fig. 2. Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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

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

3. ASYNCHRONOUS ADAPTIVE SPECIALIZATION

In order to reconcile software engineering principles for generality with performance demands 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.

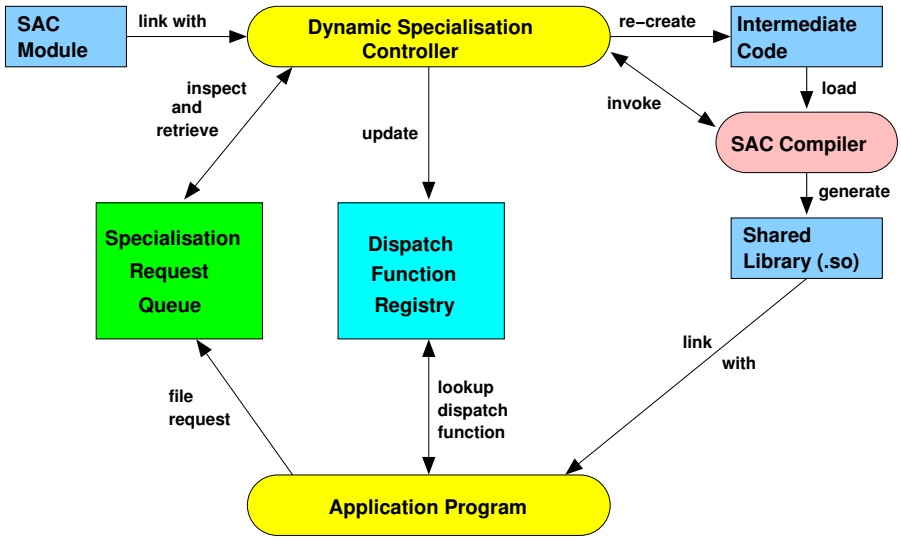


Fig. 3. Software architecture of asynchronous adaptive specialization framework

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.

In [Grelck et al. 2012] we validate our approach on a reasonably simple benchmark code: generic convolution with convergence check, as shown in Fig. 4 (A detailed explanation of the code can be found in [Grelck 2012]). In this scenario we alternately execute two functions subject to adaptive specialization: a single convolution step and the convergence criterion. (Note that we could alternatively have specialized the function `convolution` in one step, but we chose the above scenario for the purpose of illus-

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

Fig. 4. Case study: generic convolution in SAC (simplified excerpt)

tration.) Fig. 5 demonstrates how the execution time per iteration, consisting of one convolution step and one evaluation of the convergence test, significantly shrinks in two steps when the two corresponding specialized implementations become available one after the other.

The effectiveness of asynchronous adaptive specialization crucially depends on how quickly specialized variants of critical functions become available to the running application. This 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. Execution time and compile time of any code are unrelated with each other and, thus, many scenarios are possible. Even the first specialization may only become available after the termination of the application program, or the second iteration may already benefit from optimized implementations. We do, however, expect typical scenarios to yield a fixed point after a certain, generally not too large number of specializations. If an application does not meet this restriction, we must avoid further dynamic specialization at some point.

4. MANIFOLD 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 individual specialization requests to one combined request prior to servicing it.

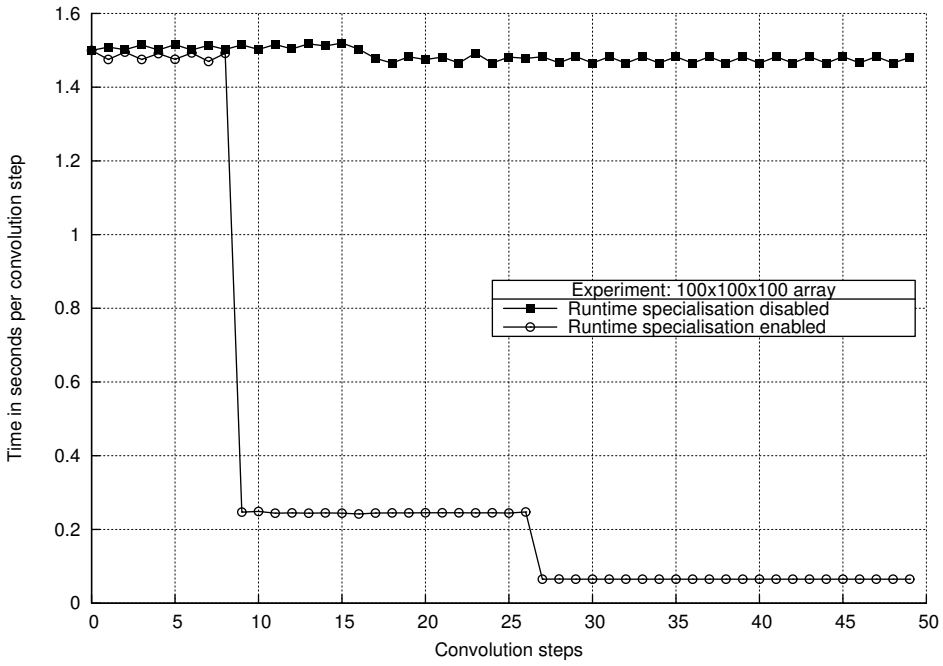


Fig. 5. Case study: running the generic convolution kernel with convergence check on a 3-dimensional argument array of shape $100 \times 100 \times 100$ with and without asynchronous adaptive specialization

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 user specific modules or the extensive SAC standard library. Many of these symbols are inline functions whose precompiled intermediate code is 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 often 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 specializations 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 example discussed in Section 3, we must admit that at least a straightforward realization of this approach does not provide 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 also the final one.

Of course, we can expect from less simplistic applications that when the specialization controller completes the first specialization, it will indeed find a request queue with multiple entries. Notwithstanding, our example application highlights 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 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.

5. PRIORITIZED ASYNCHRONOUS ADAPTIVE SPECIALIZATION

Our second approach, like the first, is concerned with the way 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 dynamically 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.

6. 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 at Fig. 5 shows that the relative performance improvements realized by adaptive asynchronous specialization by far outweigh potential improvements through data-parallel execution even when assuming linear speedups and even on the fairly small quad-core system that we used for our experiments. On system with tens of cores parallel specialization through multiple concurrent specialization controllers should be beneficial even if the relative performance improvements are less impressive.

For our running example of generic convolution it is fairly clear that two specialization controllers would be optimal. One would then specialize the convolution step while the other could concurrently specialize the convergence check. According to Fig. 5 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).

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.

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 the running example this question seems to be straightforward to answer: two. However, even for this admittedly simple demo application this is not the optimal number. Once both 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 experimentation.

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 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.

7. PERSISTENT ASYNCHRONOUS ADAPTIVE SPECIALIZATION

Another major 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 termination. Consequently, any follow-up run of the same application program starts again from scratch. Of course, the next run may use arrays of different shape, but in many scenarios it is quite likely that a similar set of shapes will prevail as in previous runs.

Therefore, we propose to store specialized binary functions in persistent collections alongside the original generic binary modules. 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 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 results in a sort of training phase in practice after which most required specializations have become available. Only in case the user runs an application on a previously unseen 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. 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 when which specializations may be displaced by new ones. In other words, persistent storage is managed like a cache memory for specializations.

8. RELATED WORK

Our approach is related to a plethora of work in the area of just-in-time compilation; for a survey see [Aycock 2003]. 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 [Aycock 2003]. 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 [Streit et al. 2012], 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 [Kim et al. 2007] (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. This makes our approach more versatile and hardware independent, but might grant COBRA the better optimization potential.

Another project with architecture similarities to our proposed system is Jikes RVM [Arnold et al. 2000; 2005]. 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 and installs the resulting optimized method into the VM. While this

architecture matches our system already quite closely, the optimizations performed are still platform oriented.

Other notably similar systems include ADAPT [Voss and Eigenmann 2000; 2001], a system that uses a domain specific language to specify optimization hints that can be made use of at runtime, ADORE [Lu et al. 2003], a predecessor of COBRA for single threaded applications, and earlier work on an adaptive runtime optimization system for FORTRAN [Hansen 1974].

Some of the research directions that we have proposed in this paper have been investigated in other dynamic compilation scenarios quite different from our's of high-level array programming, e.g. caching of specialized code blocks [Nohl et al.] or parallel dynamic compilation [Qin et al. 2006; Böhm et al. 2011]. While being motivated by the same goal, reducing the effective cost of adaptive specialization / dynamic compilation, the concrete work differs considerably from our's.

9. CONCLUSIONS AND FUTURE WORK

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.

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 sooner. 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. 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. Last not least, combining multiple specialization requests in one dynamic compilation process helps to reduce the effective overhead per specialization, and consciously selecting the most promising specialization requests aims at making the best possible use of the available computing resources.

We are currently 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.

REFERENCES

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, , and Peter F. Sweeney. 2000. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00), Minneapolis, USA*. ACM.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, , and Peter F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (2005).

John Aycock. 2003. A Brief History of Just-In-Time. *Comput. Surveys* 35, 2 (2003), 97–113.

Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized Just-in-time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, USA, 74–85. DOI:http://dx.doi.org/10.1145/1993498.1993508

M. Diogo and C. Grelck. 2013. Towards Heterogeneous Computing without Heterogeneous Programming. In *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK (Lecture Notes in Computer Science)*, K. Hammond and H.W. Loidl (Eds.). Springer.

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

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

C. Grelck. 2012. Single Assignment C (SAC): High Productivity meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.), Vol. 7241. Springer, 207–278.

Clemens Grelck and Sven-Bodo Scholz. 2003. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters* 13, 3 (2003), 401–412.

Clemens Grelck and Sven-Bodo Scholz. 2006a. Merging Compositions of Array Skeletons in SAC. *Journal of Parallel Computing* 32, 7+8 (2006), 507–522.

Clemens Grelck and Sven-Bodo Scholz. 2006b. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427.

Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. 2010. 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.

Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. 2012. Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. *Concurrency and Computation: Practice and Experience* 24, 5 (2012), 499–516.

Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. 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, 15–24.

G.J. Hansen. 1974. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, USA.

International Standards Organization. 1993. *Programming Language APL, Extended*. ISO N93.03. ISO.

K.E. Iverson. 1991. *Programming in J*. Iverson Software Inc., Toronto, Canada.

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

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

Jinpyo Kim, Wei-Chung Hsu, and Pen-Chung Yew. 2007. COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications. In *International Conference on Parallel Processing (ICPP 2007)*.

Dietmar Kreye. 2002. A Compilation Scheme for a Hierarchy of Array Types. In *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers (Lecture Notes in Computer Science)*, Thomas Arts and Markus Mohnen (Eds.), Vol. 2312. Springer, 18–35.

Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. 2003. 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.

A. Nohl, G. Braun, O. Schliebusch, and others. A universal technique for fast and flexible instruction-set architecture simulation. In *39th Design Automation Conference (DAC'02), New York, NY*. ACM.

Q. Qin, J. D'Enrico, and X. Zhu. 2006. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)*. ACM, 193–198.

Kevin Streit, Clemens Hammacher, Andreas Zeller, , and Sebastian Hack. 2012. Sambamba: A Runtime System for Online Adaptive Parallelization. In *21st International Conference on Compiler Construction (CC'12), Tallinn, Estonia (Lecture Notes in Computer Science)*, M. O'Boyle (Ed.), Vol. 7210. Springer, 240–243.

M.J. Voss and R. Eigenmann. 2000. A Framework for Remote Dynamic Program Optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00), Boston, USA*. ACM, 32–40.

M.J. Voss and R. Eigenmann. 2001. High-Level Adaptive Program Optimization with ADAPT. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, USA*. ACM, 93–102.