Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming and Beyond

Grelck, C.

Publication date
2011
Document Version
Final published version
Published in
Arbeitsberichte des Instituts für Wirtschaftsinformatik

Westfälische Wilhelms-Universität Münster

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgeber: Prof. Dr. J. Becker, Prof. em. Dr. H. L. Grob,
Prof. Dr.-Ing. B. Hellingrath, Prof. Dr. S. Klein, Prof. Dr. H. Kuchen,
Prof. Dr. U. Müller-Funk, Prof. Dr. G. Vossen

Arbeitsbericht Nr. 132

Tagungsband 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11)

26. bis 28. September 2011, Schloss Raesfeld, Münsterland

Herbert Kuchen, Tim A. Majchrzak, Markus Müller-Olm (Hrsg.)

ISSN 1438-3985

Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming and Beyond

Clemens Grelck

University of Amsterdam
Institute of Informatics
Science Park 904
1098 XH Amsterdam
Netherlands
c.grelck@uva.nl

Abstract. We present the concept of an adaptive compiler optimisation framework for the functional array programming language SaC, Single Assignment C. SaC advocates shape- and rank-generic programming with multidimensional arrays. A sophisticated, highly optimising compiler technology nonetheless achieves competitive runtime performance on a variety of sequential and parallel computer systems. This technology, however, is based on aggressive specialisation of rank-generic code to concrete array shapes, which is bound to fail if no information about potential array shapes is available in the code.
To reconcile abstract, rank- and shape-generic programming with high demands on runtime performance under all circumstances, we blur the distinction of compile time and runtime and augment compiled application programs with the ability to dynamically adapt their own executable code to the concrete ranks and shapes of the data being processed. For this adaptation we make use of our heavy-weight, non-just-in-time, but highly optimising compiler. To avoid delays in program execution we build on modern multi-core hardware and run any re-compilation process asynchronously with the application that triggers it.

1 Introduction

Functional array programming, as advocated by the language SaC [1, 2], replaces lists and trees by multidimensional arrays as prevalent data organisation principle. Such arrays are regular collections of uniform data with two essential structural properties: *rank* and *shape*. The *rank* of an array is a natural number that describes the number of dimensions or axes of an array. The *shape* of an array is a vector of natural numbers, whose length equals the array's rank. It describes the extent of the array, more precisely the number of elements alongside each dimension or axis.

This notion of multidimensional arrays naturally induces a three-level structural subtype relationship for any element data type. At the lowest level, both rank and shape are known to the compiler, e.g. we deal with a 100×100 matrix

(rank: 2, shape: [100,100]). At an intermediate level we may still know the rank of an array at compile time, but not the concrete shape, e.g. a matrix (rank: 2). Last not least, at the most abstract level neither the shape nor the rank may be available up until runtime. In principle, this subtyping hierarchy could be extended to partial static shape information, but we refrain from this for now and stick to the three layers as described above. While our subtyping hierarchy has exactly three layers on the vertical axis, it is unbounded on the horizontal axis: there is, for instance, an infinite number of different matrices. There are likewise infinitely many different array ranks, although we admit that in practice only a fairly small number of different array ranks is relevant.

Functional array programming exposes a very common software engineering dilemma in a nutshell. We naturally aim at describing any problem at the highest possible level of abstraction with respect to our subtyping hierarchy. For example, we want to define matrix multiplication in a shape-generic way, i.e. to be applicable to any matrix, not a matrix of 100×100 elements. Furthermore, we prefer to define basic array operations, such as rotation, only once in a rank-generic way. Rank- and shape-generic programs foster code reuse and lead to very concise algorithmic specifications.

The above mentioned dilemma stems from the fact that we are also concerned with runtime performance. To this effect it is not so difficult to see that abstraction in specifications induces cost in execution. There is a plethora of reasons for this: compiler transformations are hindered by the absence of static information, generic arrays must carry around and maintain their structural properties, to name just a few.

Compiler optimisations aim at overcoming this dilemma and reconcile software engineering demands for generic specifications with user demands for high runtime performance. Automatic specialisation of rank- and shape-generic code to concrete ranks and shapes plays a crucial role in the compilation of functional array programs. Specialisation is motivated by a common observation. While the number of instances is unbounded both on the rank- and on the shape-generic level of our subtyping hierarchy, the number of different instances effectively occurring in some program run is typically very small. Take image processing as an example. While image processing algorithms should certainly abstract from the concrete size (shape) of argument images, the number of actually used image formats is fairly small.

Unfortunately, automatic specialisation by the compiler is not the ultima ratio. One problem is, of course, that specialisation increases object code sizes, but we currently do not consider this to be a show-stopper in other than pathological cases. The real show-stopper is the potential, and in practice quite common, inability of a compiler to determine the relevant shapes and ranks for specialisation. Program code may simply be too obfuscated for a compiler to figure out the relationships between array shapes and ranks properly. Moreover, and more fundamentally, program code may not contain any useful hint as to what ranks and shapes may become relevant at runtime. To stick to our image processing example, the concrete size of argument images may only be determined by the

concrete images on a user's disk. As a matter of principle, it may be unknown at compile time of an application to what shapes (and to a lesser extent ranks) program code will be applied by some user. This is a principle problem that cannot be overcome by more sophisticated static compiler analysis.

In SAC we approach this issue with a novel adaptive re-compilation framework that is the subject of this paper. Whenever program execution reaches a suitable shape- or rank-generic function we on-the-fly generate code specialised to the concrete shapes of the various argument arrays and dynamically link this new code into the running application. By doing so we incrementally adapt the running application to the shapes and ranks relevant in the user's application domain.

A few aspects set our work apart from other just-in-time compilation approaches. Firstly, the SAC compiler is not a lean, tailor-made just-in-time compiler. It rather takes its strength from sophisticated code analyses and far-reaching code transformations [3, 4]. Consequently, re-compilation is costly, but the cost is inevitable to achieve the desired performance levels. To overcome this dilemma, our re-compilation framework exploits recent advances in computer architecture. With increasing core counts even on general-purpose processors not all cores can effectively be exploited through automatic parallelisation [5] of application code, even in the case of SAC.

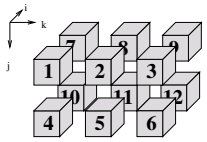
Even with very satisfactory speedup characteristics using 14 or 16 cores for an application program makes little difference in effective program execution times. Hence, the idea is to set apart one or a few of the available cores on the execution machinery for runtime adaptive re-compilation. In other words, re-compilation is fully asynchronous with the running application itself and thus inflicts no measurable overhead, but the next time a generic function is applied to arguments with the same structural properties the user will benefit from the improved runtime behaviour of the specialised version. As pointed out before, the entire approach is based on the assumption that some generic function is indeed repeatedly applied to argument arrays with identical shapes. While this is to a certain degree speculative, many applications indeed expose this characteristic.

The remainder of the paper is organised as follows. Section 2 provides a few more details on the design of SAC. In Section 3 we illustrate generic functional array programming in SAC by means of a small example, before we present our ideas on adaptive compilation in more detail in Section 4 and show some experimental results in Section 5. We sketch out directions of on-going work in Section 6 and draw conclusions in Section 7.

2 SAC in a Nutshell

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 is meant to facilitate familiarisation for programmers who rather have a background in imperative languages than in declarative languages. Core SAC is a functional, side-effect free subset of C: we interpret assignment sequences as

nested let-expressions, branching constructs as conditional expressions and loops as syntactic sugar for tail-end recursive functions. Details on the design of SaC and the functional interpretation of imperative-looking code can be found in [1]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation [5].



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

On top of this language kernel SaC provides genuine support for truly multidimensional and truly stateless/functional arrays advocating a shape- and rank-generic style of programming. Conceptually, any SaC expression denotes an array; arrays can be passed to and from functions call-by-value. A multidimensional array in SaC is represented by a *rank scalar* defining the length of the *shape vector*. The elements of the shape vector define the extent of the array along each dimension and the product of its elements defines the length of the *data vector* The data vector contains the array elements (in row-major order). Fig. 1 shows a few examples for illustration. Notably, the underlying array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Furthermore, we achieve a complete separation between data assembled in an array and the structural information (rank and shape).

The type system of SaC (at the moment) is monomorphic in the element type of an array, but polymorphic in the structure of arrays, i.e. rank and shape. As illustrated in Fig. 2, each element type induces a conceptually unbounded number of array types with varying static structural restrictions on arrays. These

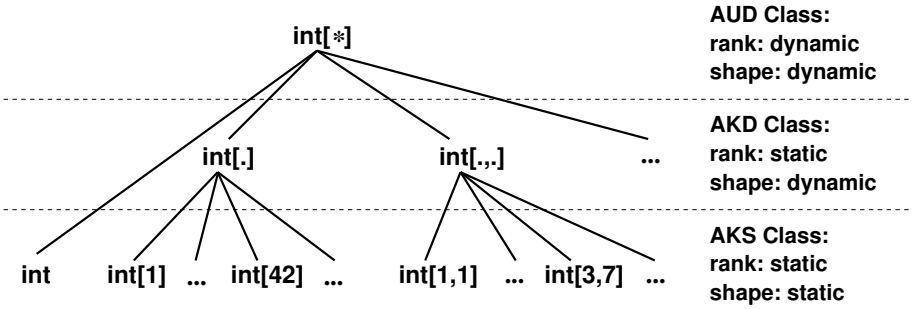


Fig. 2: Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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.

SAC only provides a small set of built-in array operations. Essentially, there are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`). A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*`[`*idxvec*`]`. The use of a vector for the purpose of indexing into an array is crucial in a rank-generic setting: if the number of dimensions of an array is left unknown at compile time, any syntax that uses a fixed number of indices (e.g. comma-separated) makes no sense whatsoever.

While simple (one-dimensional) vectors can be written as shown in Fig. 1, i.e. as a comma-separated list of expressions enclosed in square brackets, any rank- or shape-generic array is defined by means of WITH-loop expressions. The WITH-loop is a versatile SAC-specific array comprehension or map-reduce construct. Since the ins and outs of WITH-loops are not essential to know for reading the rest of the paper, we skip any detailed explanation here and refer the interested reader to [1] for a complete account.

A number of case studies have been performed over the years and illustrate the combination of high-level array programming and runtime performance on a variety of target architectures [6–9].

3 Case Study: Generic Convolution

In order to illustrate the shape-generic high-level programming style typical of SAC and, in consequence, to demonstrate the effect of the proposed adaptive compilation framework, we introduce a small case study: generic convolution

with iteration count and convergence check. This computationally non-trivial and application-wise highly relevant numerical kernel fits on half a page of SaC code; Fig. 3 contains the complete implementation.

```
1  module Convolution;
2
3  use Array: all;
4
5  export {convolution};
6
7  double[*] convolution (double[*] A, double epsilon,
8                         int max_iterations)
9  {
10   i = 0;
11
12   A_new = A;
13
14   do {
15     A_old = A_new;
16     A_new = convolution_step( A_old);
17     i += 1;
18   }
19   while (!is_convergent( A_new, A_old, epsilon)
20          && i < max_iterations);
21
22   return( A_new);
23 }
24
25 inline double[*] convolution_step (double[*] A)
26 {
27   R = A;
28
29   for (i=0; i<dim(A); i++) {
30     R += rotate( i, 1, A) + rotate( i, -1, A);
31   }
32
33   return( R / tod( 2 * dim(A) + 1));
34 }
35
36 inline bool is_convergent (double[*] new, double[*] old,
37                            double epsilon)
38 {
39   return( all( abs( new - old) < epsilon));
40 }
```

Fig. 3: Case study: generic convolution kernel with convergence check

The first line of code defines a module Convolution. This module makes intensive use of the Array module from the SaC standard library, that defines a large number of typical array operations such as array extensions of the usual scalar primitive operators and functions, structural operations like shifting and rotation or reduction operations like sum or product of array elements. The last line of the module header declares that our module Convolution exports a single symbol, i.e. the function convolution.

The function convolution is defined in lines 7–23; it expects three arguments: an array A of double precision floating point numbers of any shape and any rank, which is the array to be convolved, a double precision floating point number epsilon that defines the desired level of convergence and an integer number max_iterations that is supposed to prematurely terminate the convolution after a given number of iterations regardless of the convergence behaviour.

The body of the function convolution essentially consists of the iteration loop, a C-style do/while-loop. This nicely demonstrates the close syntactical relationship between SaC and C. Semantically, however, the SaC do/while-loop is merely syntactic sugar for an inlined tail-recursive function. Nonetheless, the SaC programmer hardly needs to reason about such subtle semantical differences as the observable runtime behaviour of the SaC code is exactly the same as one would expect from the corresponding C code.

Within the do/while-loop of function convolution we essentially perform a single convolution step that itself is implemented by the function convolution_step defined in lines 25–34. This function expects an array A of double precision floating point numbers of any shape and any rank and yields a new array of the same shape and rank. In our example, we chose cyclic boundary conditions as exemplified by the use of the rotate function from the SaC standard array library. In fact, the function rotate(index, offset, array) creates an array that has the same rank and shape as the argument array array, but with all elements rotated by offset index positions along array axis (or dimension) index. With the for-loop in lines 29–31 we rotate the argument array twice in each dimension, by one element towards decreasing and by one element towards increasing indices. Each time we combine the rotated arrays using element-wise addition, as implemented by an overloaded version of the + operator. In essence, this implements a rank-invariant direct-neighbour stencil operation, i.e., in the 1-dimensional case we have a 3-point stencil, in the 2-dimensional case a 5-point stencil, in the 3-dimensional case a 7-point stencil and so on.

In many concrete applications we will have different weights for different neighbours. In order to bound the complexity of our case study we refrain from supporting this here, albeit such an extension would be rather straightforward. Instead, we merely compute the arithmetic mean, i.e. all neighbours and the old value have the same weight. To achieve this we divide all elements of array R by the number of neighbours plus one for the old value. The function tod merely converts an integer number into a value of type double.

Coming back to the definition of the function convolution we may want to have a closer look at the loop predicate of the do/while-loop in lines 19/20. We

continue as long as we neither detect convergence nor the maximum number of iterations is reached. While the latter requires a simple comparison on integer scalar values, the former makes use of the generic convergence test defined in lines 36–40. The function is_convergent checks whether for all elements of the argument arrays new and old the absolute value of the difference is less than the given convergence threshold epsilon. This function definition is a nice example of the SaC programming methodology that advocates the implementation of new array operations by composition of existing ones. All four basic array operations used here, i.e. element-wise subtraction, element-wise absolute value, element-wise comparison with a scalar value and reduction with Boolean conjunction (all), are defined in the SaC standard library.

4 Adaptive Compilation Framework

The architecture of our adaptive compilation framework is sketched out in Fig. 4. A key design choice in our framework is the separation of the gathering of profiling information that triggers specialisation (bottom part of Fig. 4) from the actual runtime specialisation itself (upper part of Fig. 4). Our aim was to keep the implementation of the former as lean as possible to keep the impact on compiled application code minimal. Instead, most of the new functionality is encapsulated in the *dynamic specialisation controller*. The running program and the dynamic specialisation controller communicate with each other exclusively via two shared data structures: the dispatch function registry and the specialisation request queue (center of Fig. 4). This lean and well-defined interface facilitates the use of multiple specialisation controller instances on the one side and supports multithreaded execution of the program itself on the other side of the interface.

Our design makes use of the existing function call infrastructure within executable (binary) SaC programs generated by our SaC compiler `sac2c`. Such programs (generally) consist of binary versions of shape-specific, shape-generic and rank-generic functions. Any shape-generic or rank-generic function, however, is called indirectly through a *dispatch function* that selects the correct instance of the function to be executed in the presence of function overloading by the programmer and static function specialisation by the compiler. This dispatch function serves as an ideal hook to add further instances (specialisations) of functions created at runtime. Since adding more and more instances also affects function dispatch itself, we need to change the actual dispatch function each time we add further instances. To achieve this we no longer call the dispatch function directly, but do this through a pointer indirection that allows us to exchange the dispatch function dynamically as needed. We call the central switchboard that implements the function call forwarding the *dispatch function registry*.

Before actually calling the dispatch function retrieved from the registry, we file a specialisation request in the *specialisation request queue*. Apart from information that allows us to uniquely identify the target of the call, this request contains the concrete shapes of all actual arguments. It is essential here that

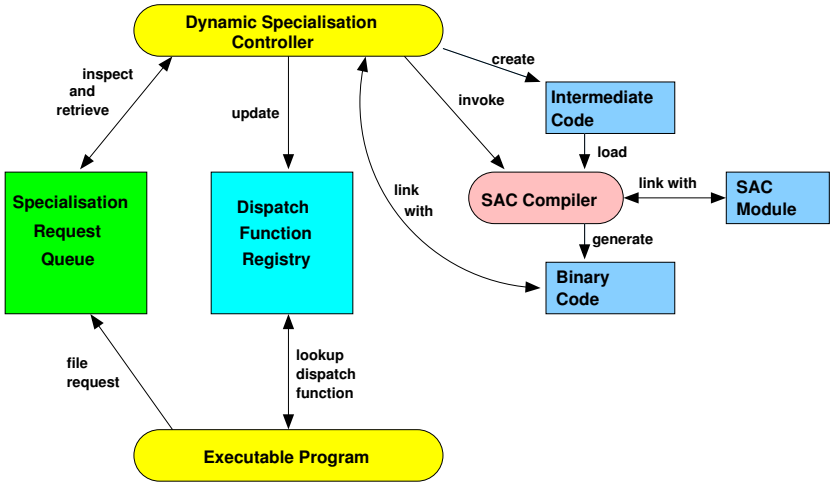


Fig. 4: Architecture of our adaptive compilation framework

queuing specialisation requests is implemented as lightweight as possible as this operation is performed for every call to a generic function. To achieve this, we have slimmed the operation down as far as possible. Most information contained in the specialisation request is precomputed and preassembled at compile time. Furthermore, we do not perform any sanity checks during the enqueue operation. These are postponed until the request is later processed by the asynchronous specialisation controller. This design is geared towards reducing the impact of the proposed adaptive compilation framework on the genuine program execution to a minimum.

Within the same process that runs the *executable program* one or more threads are set apart acting as *dynamic specialisation controllers*. A dynamic specialisation controller is in charge of the main part of the adaptive compilation infrastructure. A dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the same specialisation request has been issued before and is currently being processed. If so, the request is discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of the specialised function instance to be generated.

The dynamic specialisation controller then invokes the SAC-compiler `sac2c` on the intermediate representation, i.e. the dynamic specialisation controller effectively turns itself into the SAC-compiler. As such, it now starts the standard compilation process for the generated intermediate representation. During this process, the compiler dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SAC module

system that was originally developed to support inter-module (compile time) optimisation [10].

Eventually, the SaC-compiler (with the help of a backend C compiler) generates another shared library containing binary versions of the specialised function(s) and one or more new dispatch functions taking the new specialisations into account in their decision. Following the completion of the SaC-compiler, the dynamic specialisation controller regains control.

Before attending to the next specialisation request, two tasks still need to be performed to enable the new specialised code in the running program. Firstly, the controller links the running process with the newly created shared library. As the module name chosen for the stub is unique, this will make a new symbol for the dispatch code of the specialised function available. In a second step, the controller updates the dispatch function registry with the new address of this new symbol for dispatch function(s) from the newly compiled library. As a consequence, any subsequent call to the now specialised function originating from the running program will directly be forwarded to the specialised instance rather than to the generic version and benefit from (potentially) substantially higher runtime performance without further overhead.

5 Experimental Evaluation

Our experimental evaluation is based on the generic programming case study introduced in Section 3. We use an AMD Phenom II X4 965 quad-core system running at 3.4GHz clock frequency. It is equipped with 4GB DDR3 memory, and the operating system is Linux with kernel 2.6.38-rc1.

Fig. 5 shows recorded execution times for convolution of a 1000×1000-matrix. We uniformly set the number of iterations to 50 while we use a convergence threshold and initial array values which guarantee that we effectively run these 50 iterations. Since we read some initialisation data from a file, the SaC compiler is unable to deduce this information statically, and we effectively evaluate the convergence check in every iteration. To isolate the effect of adaptive compilation more clearly, we refrain from running the application itself with multiple threads for now and only employ a single instance of the specialisation controller throughout the experiments.

In Fig. 5 we can see that at the beginning, code with and without runtime specialisation enabled takes about the same time per iteration. We can observe a small overhead for the version with runtime specialisation which stems from the filing of specialisation requests and the checking of the dispatch function registry.

In our case study, the first convolution iteration triggers the first specialisation requests for functions `convolution_step` and `is_convergent`. As soon as specialised and, in consequence, far better optimised versions of these functions become available we can identify a dramatic decrease in single iteration runtimes. As the problem size remains constant throughout each individual program run, no further specialisations occur and the shorter iteration runtime remains con-

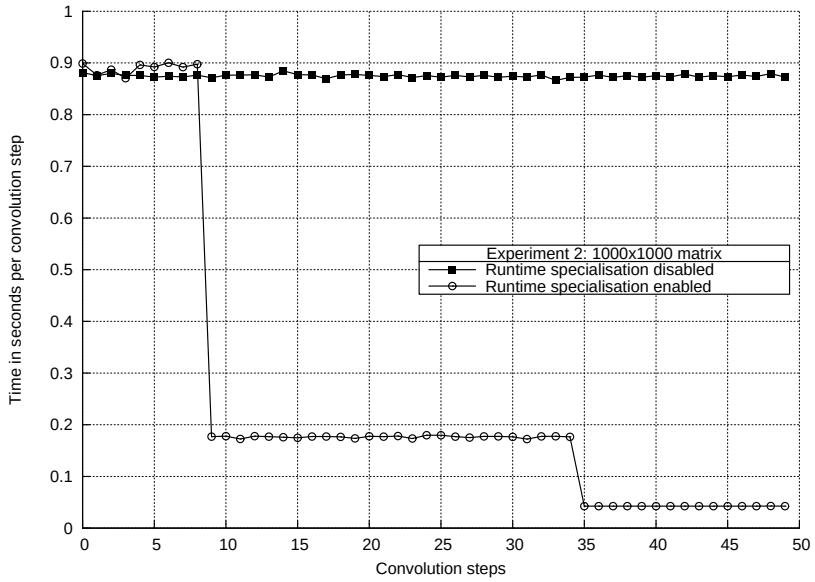


Fig. 5: Experimental results obtained by applying the case study code discussed in Section 3 to a 1000×1000-matrix with runtime specialisation disabled and enabled

stant until termination. The time it takes to dynamically recompile versions of the functions convolution_step and is_convergent specifically adapted to the individual experimental settings is, of course, independent of these settings in general and the problem size used in particular.

6 Beyond a Single Program Run

For now, we only collect specialisations during one program run. As soon as an application terminates, all dynamic specialisations created are discarded. However, it is by no means unlikely that in a following invocation of the same application also the same specialisations are required and, as a consequence, regenerated. This observation is also not limited to a single application. Many different applications often share large parts of the code in intermediate abstraction layers. Last not least, the SaC standard library is the common basis for almost any SaC application.

In fact, the same specialisations are likely to be helpful across multiple invocations of the same program, across different programs and beyond a single user. Hence, we aim at persistently storing specialised binary code alongside the generic binary code of an original module implementation and not temporarily alongside some application binary. Thus, we would over time create a growing base of pre-specialised instances of certain functions ready for use in subsequent

program invocations, including new programs. As a consequence, the startup overhead that we can observe in Section 5, both from generating the necessary specialisations and from executing alternative generic code, would often be reduced to nothing if the necessary specialisations can be obtained from some repository right away.

The administration of large numbers of persistent function specialisations on the level of a SaC installation and beyond the realm of a single application or a single user raises many interesting and challenging further problems. It goes without saying that such specialised function repositories cannot continuously grow. Instead we need to adopt certain caching strategies in conjunction with a bounded repository size that could for instance be determined upon system installation. Strategies such as least-recently-used, maybe in conjunction with more complex usage statistics, come to mind.

7 Conclusion

We have presented an adaptive compilation framework for generic functional array programming that virtually achieves the quadrature of the circle: to program code in a generic, reuse-oriented way abstracting from concrete structural properties of the arrays involved, and, at the same time, to enjoy the runtime performance characteristics of highly specialised code when it comes to program execution.

As multiple cores are already rather the norm in contemporary processors, and the number of cores is predicted to grow quickly in the near future, adaptive optimisation virtually comes for free. We run all dynamic recompilations/specialisations fully asynchronously with the main computation. Thus, their delaying effect on the main computation is minimal. With the growing numbers of cores this observation even holds for computational code that is itself multithreaded. Reserving a small fraction of available cores for adaptive compilation either permanently or temporarily has a minor effect on the computation's progress even for linearly scaling programs.

Since each dynamic invocation of the compiler incurs some overhead independent of the problem size, adaptive compilation becomes more profitable for long-running applications. In an extreme case, the program itself could run to completion before the first spawned specialisation request is actually satisfied. On the other extreme end of the spectrum an application an application may likewise run fully specialised code for almost all of the application runtime. In the worst case, the main wasting of resources would be in occupying one core to produce code that is never going to be run. However, many numerically interesting/challenging real-world problems are indeed long-running relative to compilation times.

Likewise, our approach builds on the assumption of temporal locality, i.e., the assumption that if some function is applied to some arguments of certain shapes it will later during program execution again be applied to arguments of the same shapes (but most likely different values). If this assumption does not hold

for a certain program, adaptive compilation cannot be expected to provide any benefits to runtime performance. In such a case it should rather be disabled to avoid wasting resources such as the core used for program adaptation. However, the detrimental effect of adaptive compilation on the performance of the program itself is very small.

Acknowledgements

This work is supported by the European Commission through the FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through Performance Analysis to Support Concurrency Engineering), grant no. FP7 248828.

A more detailed account of the design and implementation of SAC's adaptive specialisation and optimisation system can be found in [11].

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
2. Grelck, C., Scholz, S.B.: SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In Glew, N., Blelloch, G., eds.: Annual Symposium on Principles of Programming Languages, Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France, ACM Press, New York City, New York, USA (2007) 25–33
3. Grelck, C., Scholz, S.B.: SAC — From High-level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13** (2003) 401–412
4. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
5. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
6. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press (2002)
7. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In Malyshkin, V., ed.: Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. Volume 2763 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2003) 230–235
8. Shafarenko, A., Scholz, S.B., Herhut, S., Grelck, C., Trojahner, K.: Implementing a Numerical Solution of the KPI Equation using Single Assignment C: Lessons and Experiences. In Butterfield, A., ed.: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05). Dublin, Ireland, September 19–21, 2005, Revised Selected Papers. Volume 4015 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Heidelberg, New York (2006) 160–177
9. Kudryavtsev, A., Rolls, D., Scholz, S.B., Shafarenko, A.: Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In: 10th International Conference on Parallel Computing Technologies (PaCT'09). Volume 5083

of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Heidelberg, New York (2009) 445–456
10. Herhut, S., Scholz, S.B.: Towards Fully Controlled Overloading Across Module Boundaries. In Grelck, C., Huch, F., eds.: 16th International Workshop on the Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany, University of Kiel (2004) 395–408
11. Grelck, C., van Deurzen, T., Herhut, S., Scholz, S.B.: Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. Concurrency and Computation: Practice and Experience (2011)