An Adaptive Compilation Framework for Data-Parallel Array Programming in SAC

Clemens Grelck, Tim van Deurzen
University of Amsterdam
Institute of Informatics, Amsterdam, Netherlands
{C.Grelck,Timothy.vanDeurzen}@uva.nl

Stephan Herhut, Sven-Bodo Scholz
University of Hertfordshire
School of Computer Science, Hatfield, United Kingdom
{S.A.Herhut,S.Scholz}@herts.ac.uk

SAC advocates shape- and rank-generic programming on multi-dimensional arrays, i.e. SAC supports functions that abstract from the concrete shape (extent along dimensions) and even from the rank (number of dimensions) of argument arrays [4]. A multidimensional array in SAC is represented by a triple consisting of the *rank scalar* that defines 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 determines the length of the *data vector*, which contains the array elements (in row-major unrolling). Depending on the amount of compile time structural information we distinguish between three classes of arrays at runtime: For rank-generic arrays, all three properties (i.e. rank scalar, shape vector and data vector) are variable. For shape-generic arrays, the rank scalar is a compile time constant: The length of the shape vector is known in advance, but its elements are not. For non-generic arrays both rank scalar and shape vector are constants.

From a software engineering point of view it is (almost) always desirable to specify functions on the most general input type(s) to maximise 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.

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 substantially lower for rank-generic code [6]. The reasons are manifold and their individual impact operation-specific, but three categories can be identified nevertheless: 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 optimisations [2, 3] are not as effective on generic code because certain properties that trigger program transformations cannot be inferred. Thirdly, in automatically parallelised code [1] many organisational decisions must be postponed until runtime and the ineffectiveness of optimisations inflicts frequent synchronisation barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC compiler aggressively specialises rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analyses for rank and shape specialisation, this approach is fruitless if the necessary information is not available at compile time as a matter of principle because, for example, the corresponding data is read from a file or the SAC code is called from external (non-SAC) code via the sac4c foreign language interface [7]. The latter becomes more and more important as SAC is used as the primary component language by the coordination language S-Net [5].

To mitigate the negative effect of generic code on runtime performance when specialisation is not an option, we propose an adaptive compilation framework that postpones specialisation until runtime time when all shape information is eventually available. Our idea goes as follows. Let us assume we compile a SAC library of generic functions for external use, i.e. there is no opportunity for rank or shape specialisation whatsoever. Instead of generating (utterly) inefficient generic code to be called from a host program, we generate actually two codes per function: one that reinstantiates an intermediate (compiler internal) representation of the function and one to be called from the host program as a proxy.

A host program calls the proxy function. The proxy function dynamically links the running code with the shared library containing the reinstantiation code and runs that code. As a result the running code now has access to an intermediate representation of the function and to the concrete shapes of argument arrays. It combines these two aspects to augment the reinstantiated intermdiate generic code with appropriate specialisation information. After that, the proxy function dynamically links the running code with (a variant of) the SAC compiler and runs the compiler on the dynamically created intermediate code. With all shape information available, the SAC compiler can easily specialise the rank or shpe generic code to non-generic code and draw from its complete optimisation potential to generate efficient (parallel) code for the function. A standard C compiler is used to generate executable code for the current platform, more precisely another shared library. The proxy function called by the host program eventually links with that shared library as well and applies the most efficient code to the given arguments.

We plan two refinements: Any adapted function goes into a repository. Whenever the same proxy function is called again with structurally equivalent arguments, the adpated function is called immediately rather than being re-adapted from generic code. The second refinement is more of a speculative nature and motivated by the expected abundance of computing resources in the near future. The proxy function could actually run the dynamic recompilation code as described above and the original (inefficient) generic code concurrently. If the generic code is still faster than recompilation and running of the non-generic code, we can produce a result quicker. Nevertheless, the specialised version once available will end up in the repository and may speed up future invocations of the proxy function. Over time our approach leads to a growing selection of highly optimised specialisations of generic functions that incrementally adapt to the structural properties of argument arrays of library functions. Experience has it that it is quite typical for array programs to effectively use a limited number of array shapes even if concrete shapes are not known until runtime.

Our approach differs from just-in-time compilation of Java byte code (or similar) in several aspects. Hot spots of byte code are adapted to the platform they run on by generating native code while the execution platform was left open deliberately at compile time. Adaptation of (byte) code to its execution environment happens in a single step. In contrast, our approach adapts code not to its execution environment but to the structural properties of array arguments, i.e. the data our code operates on. This adaptation is an incremental process over the execution time of a program. If the number of different array shapes used is bound, this adaptation process converges to a fixed point.

The differences to just-in-time compilation are not just qualitative but also quantitative, partially rooted in application characteristics of array processing. We expect adapted code to run by orders of magnitude faster than generic code. And, we expect array programs to run for a long time and to repeatedly execute the same functions on a relatively small set of argument ranks and shapes. Under these assumptions we are confident that the benefits of our approach amortise the invocation of a fully-fledged optimising compiler at runtime. We also believe that our findings can be generalised to other languages as the principles are independent of the concrete design of SAC.

References

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

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

[3] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, vol. 32(7+8), pp. 507–522, 2006.

[4] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, vol. 34(4), pp. 383–427, 2006.

[5] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, vol. 38(1), pp. 38–67, 2010.

[6] Dietmar Kreye. A Compilation Scheme for a Hierarchy of Array Types. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers*, *Lecture Notes in Computer Science*, vol. 2312, pp. 18–35. Springer-Verlag, Berlin, Germany, 2002.

[7] Nico Marcussen-Wulff and Sven-Bodo Scholz. On Interfacing SAC Modules with C Programs. In Markus Mohnen and Pieter Koopman, editors, *12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany*, *Aachener Informatik-Berichte*, vol. AIB-00-7, pp. 381–386. Technical University of Aachen, 2000.