An Adaptive Compilation Framework for Generic Data-Parallel Array Programming*

Clemens Grelck[1], Tim van Deurzen[1], Stephan Herhut[2], and Sven-Bodo Scholz[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck,t.vdeurzen@uva.nl
[2] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{s.a.herhut,s.scholz}@herts.ac.uk

Abstract. Generic array programming abstracts from structural properties of arrays, such as rank (number of axes/dimensions) and shape (number of element along each axis/dimension). This allows for abstract program specifications and, as such, is desirable from a software engineering perspective. However, generic programming in this sense does have an adverse effect on runtime performance, at least when executed naively. Static compiler analyses and transformations aim at reconciling software engineering desires for generic code with runtime performance requirements. However, they are bound to fail whenever the required information is not available until runtime.
We propose a compilation framework that overcomes the inherent limitations of static analysis by incrementally adapting a running program to the structural properties of the arrays it operates on. This is achieved by partial recompilation of code at runtime, when all structural properties of arrays are known, and dynamic relinking of the running program with dynamically generated code. We sketch out the general compilation framework architecture and provide some details on implementation issues.

1 Introduction

Optimising compilers reconcile the programmer's desire for generic, re-usable programs adhering to software engineering principles such as abstraction and composition and the necessities of executable code to achieve high runtime performance in sequential and, increasingly important, (implicitly) parallel execution. Optimising compilers analyse program code and infer static properties that trigger program transformations as appropriate.

The effectiveness of static analysis, however, is essentially limited by two aspects: Firstly, the quality of the analyses implemented in the compiler; and

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

secondly, the availability of required information compile time. As an example of a common compiler optimisation consider loop unrolling. Loop unrolling is triggered by a compiler analysis that infers the trip count of the loop. The compiler then unrolls the loop if the trip count is below a given threshold. However, even the best static analysis is bound to fail if the expression defining the trip count depends on values that are unknown at compile time. For example, they could be obtained from the execution environment at runtime (input), or they may be determined by code located in a different compilation unit.

We propose an adaptive compilation framework for the data-parallel functional array language SaC [1]. SaC advocates shape- and rank-generic programming on multidimensional arrays, i.e. SaC supports functions that abstract from the concrete shape (extent along dimensions) and even from the concrete rank (number of dimensions) of argument arrays and that yield result arrays whose shape and (!) rank are determined by the function itself. Depending on the amount of compile time structural information the type system of SaC distinguishes three classes of arrays at runtime:

- Arrays of Known Shape (AKS)
  where both rank and shape are statically available;
- Arrays of Known Dimensionality (AKD)
  where the rank is statically available, but the concrete shape is computed dynamically; and
- Arrays of Unknown Dimensionality (AUD)
  where neither rank nor shape are known to the compiler.

We also call these arrays non-generic, shape-generic and rank-generic, respectively.

From a software engineering point of view it is (usually) desirable to specify functions on the most general input type(s) to maximise opportunities for code reuse. Typical examples for rank-generic operations are extensions of scalar operators (arithmetic, logical, relational, etc) to entire arrays in an element-wise way or common structural operations like shifting and rotation along one or multiple axes of an array. In fact, rank-generic functions prevail in the extensive SaC standard library.

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 [2]. The reasons are manifold and their individual impact operation-specific, but three categories can be identified notwithstanding: 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 multi-dimensional array operation. Secondly, many of the SaC-compiler's advanced optimisations [3, 4] are just not as effective for generic code because the necessary code properties to trigger certain program transformations simply cannot be inferred. Thirdly, in automatically parallelised code [5] many organisational decisions must be postponed until runtime and the ineffectiveness of optimisa-

tions lead to excessive numbers of synchronisation barriers as well as 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 the necessary rank and shape information is simply not available at compile time for whatever reason. Data may be read from a file at runtime, or SaC code is called externally from a non-SaC environment via the sac4c foreign language interface [6]. In particular the latter is more and more common as we use SaC in conjunction with the component-based coordination language S-Net [7].

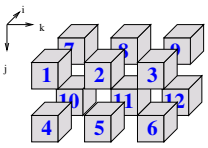
To mitigate the negative effect of generic code on runtime performance where specialisation is not an option for one or more of the aforementioned reasons, we propose an adaptive compilation framework that incrementally adapts shape- and rank-generic code to the concrete shapes and ranks used in a specific program instantiation. Our approach is motivated by the observation that the number of different array shapes that effectively appear in generic array code, although theoretically unbounded, often is relatively small in practice.

What sets our adaptive compilation framework apart from existing just-in-time compilation and dynamic optimisation/code tuning approaches is twofold. Firstly, we dynamically adapt generic code to structural properties of the data it operates on, whereas just-in-time compilation of byte code (or similar) aims at adapting code to the execution environment, e.g. by generating native machine code. The second and probably more far-reaching difference is that we inherently assume a multicore execution environment where computing resources are available in abundance and can often not completely exploited by a running program in an efficient way. Although the SaC-compiler is equipped with very effective implicit parallelisation technology [5], experience says that the difference between using 14 cores of a 16-core machine and using all cores for running a given program is often marginal because the additional overhead for organising parallel execution more and more outweighs the benefit with each core joining in into collaborative execution. At this point we propose to set apart a small (configurable) number of cores for the purpose of incrementally adapting the binary code base to the array shapes actually appearing during a program run. Our approach takes dynamic recompilation out of the critical path of an application. This property is instrumental in using a heavy-weight, highly optimising compiler like `sac2c` in an online setting.

The remainder of the paper is organised as follows. Section 2 provides a few more details on the design of Single Assignment C. We present our ideas on adaptive compilation in more detail in Section 3 and discuss implementation issues in Section 4. Eventually, we browse through related work in Section 5 and draw conclusions in Section 6.

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 processing 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).

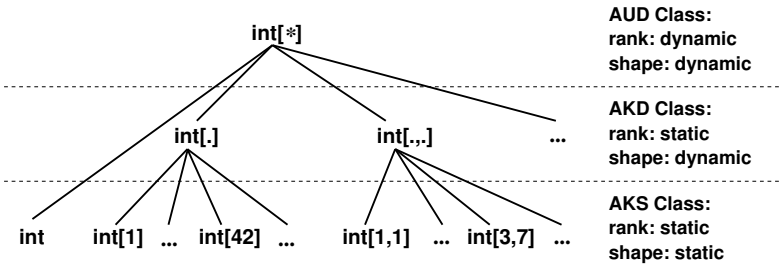


Fig. 2: Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

The type system of SaC (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 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 just like in C and other C-inspired languages, 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. In fact, 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.

3 Adaptive Compilation Framework

The architecture of our adaptive compilation framework is sketched out in Fig. 3. On the bottom of the figure we have an executable (binary) SAC program generated by our SAC compiler `sac2c`. It (generally) consists 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 whenever we add further instances. To achieve this we no longer call the dispatch function directly, but through a pointer indirection that allows us to exchange the dispatch function dynamically. We call this the *dispatch function registry*.

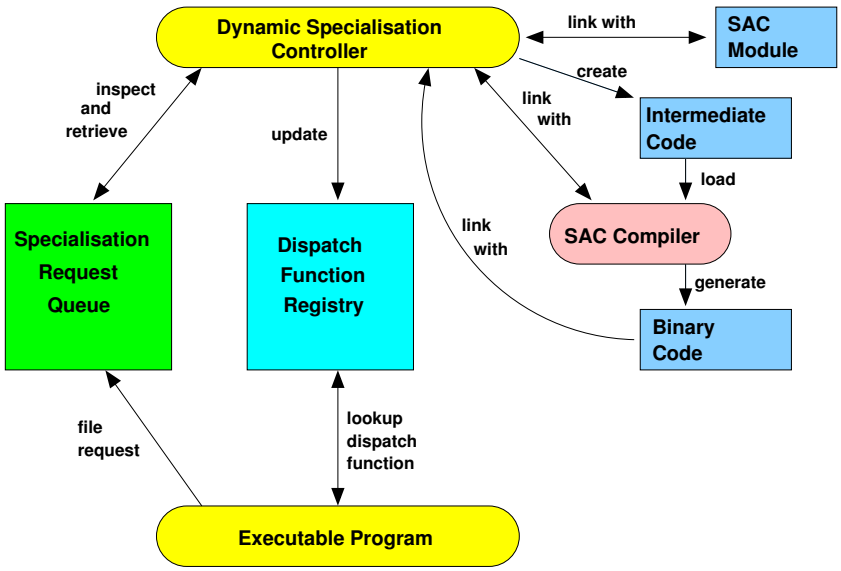


Fig. 3: Architecture of our adaptive compilation framework

Before actually calling the dispatch function retrieved from the registry, we also file a specialisation request in the *specialisation request queue*. Next to the

function name, the module name where the function originates from, etc, this request contains the concrete shape parameters of all generic parameters of that function. Queueing a specialisation request is a very lightweight operation. This makes sure that the original program execution is delayed by adaptive recompilation as little as possible.

In the same process that runs the *executable program* one thread is set apart to run the *dynamic specialisation controller*. This is in charge of the main part of the adaptive compilation infrastructure; it runs concurrently with the program itself. The dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the specialisation requested already exists or is currently in the process of being constructed. If so, the request is just discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of a new SaC-module. This consists among others of an import-statement of the function symbol from the original module and specialisation directive to the compiler generated from the specialisation request data.

The dynamic specialisation controller also links the binary executable with the entire SaC-compiler `sac2c`, which already comes as a shared library. So, having created the stub module, the dynamic specialisation controller effectively turns itself into the SaC-compiler. As such, it now 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 [8].

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 function taking the new specialisations into account in their decision. Following the completion of the SaC-compiler, the dynamic specialisation controller regains control. It still has two tasks to do before attending to the next specialisation request. Firstly, it links the running process with the newly created shared library. Secondly, it updates the dispatch function registry with the new dispatch function(s) from that library. As a consequence, any subsequent call to that function originating from the running program will directly be forwarded to the specialised instance rather than the generic version and benefit from (potentially) substantially higher runtime performance without further overhead.

Our adaptive compilation framework is carefully designed such that the associated runtime overhead in the executable program is minimal. Essentially, it boils down to an indirection in calling the dispatch function and the filing of a specialisation request. All the remaining work is done concurrently to the execution of the program itself by one or more dynamic specialisation controllers. Our assumption is that these run on different processors or cores and as such use resources that would otherwise remain unused or whose exploitation for running

the program itself would at most have a marginally positive effect on overall performance.

4 Implementation Aspects

For our prototype implementation, we have extended the existing SaC compiler and runtime system in three aspects. Firstly, we have modified the code generation of the compiler to provide the required profiling information to the specialisation controller. Secondly, we have implemented hooks in the compiler that allow the specialisation controller to initiate the specialisation of requested functions. And last but not least, we have implemented the specialisation controller itself as part of the SaC runtime system.

To control the collection and reporting of runtime information, we have added an additional flag to the compiler. The option `-runtimespec` will enable the required extension to code generation. The produced executable differs from standard executables in three main aspects. Firstly, we extend the dynamic dispatch code that is generated for function applications where we cannot statically determine the matching instance. Additionally to dynamically choosing the appropriate instance, the extended dispatch code communicates the actual parameter shapes found at runtime to the specialisation controller. As functions are always dispatched statically with respect to the base types of arguments, this information mainly comprises the rank and dimensionality of each argument. Furthermore, we send the index into the global registry that corresponds to the called function. This information is used two-fold: It allows us to later identify which entry in the registry to update. More importantly, however, the index can be used as a unique token to identify the function to specialise. We use this token to lookup the information that is required in the communication with the compiler.

Note here that we send that shape information blindly. In particular, we do not perform any checks on whether a specialisation is actually necessary. To keep the runtime overhead within the actual program as low as possible, we offload these checks into the specialisation controller.

Secondly, we reroute all function applications via the central register. By using the register instead of calling functions directly, we are able to dynamically rebind function applications to updated implementations. All that is required is an update to the function pointer in the registry.

Lastly, we have modified the static dispatch, as well. If no runtime specialisation is requested, we usually dispatch a function call statically as soon as we can identify a single matching instance. However, such instance could still be relatively generic. For instance, a most-generic instance might be defined for arguments of unknown dimensionality (AUD).

When using runtime specialisation, such a dispatch is not desirable. As we use the dynamic dispatch code to trigger runtime specialisation, an application that has been statically dispatched would never be optimised. Therefore, when runtime specialisation is enabled, we only dispatch a function application stati-

cally if we were able to derive full shape knowledge for the arguments and the matching instance is an exact match for those shapes. In those situations, no further specialisation would be possible.

The second work package in our implementation, the special version of the SaC compiler that creates new specialisations on the fly, turned out to require only limited implementation effort. We mainly make use of existing compiler features. The heavy lifting of creating the actual specialisations and updated dynamic dispatch code is performed by the SaC module system [8]. To allow for specialisation across module boundaries at compile time, SaC modules already contain, apart from the compiled binary, a condensed representation of the definition of each function. We reuse the same information for the creation of specialisations at runtime.

Furthermore, we use the ability of the SaC module system to extend functions from a different module by new instances, forming an updated version of the function in the current module.

Lastly, we use a language feature of the SaC-compiler, i.e. *forced specialisations*, to express the runtime specialisation request at the language level. To ensure that a function is specialised for certain argument types, the programmer can simply provide the desired function signature prepended with the keyword `specialize`. This will trigger a specialisation to that signature at compile time of the module or program that contains the `specialize` directive.

As an example for the interplay of these three features during runtime specialisation, consider the following scenario: Assume we have a function `add` that expects two arguments, yields one return value and is defined in module `Math`. We now want to specialise this function for two arguments of type `int[7]`. This can be achieved by the following regular SaC code:

```
module DynSpec1;

import Math : {add};

export : all;

specialize int[*] add( int[7] x, int[7] y);
```

First, we create a new container for the resulting extended function `add` in form of the module `DynSpec1`. Note that the name is of no importance as long it is unique. Next, we trigger the module system to load the existing instances of the function `add` from its defining module `Math` by means of an `import` directive. As we want to make the resulting instances available to the running program, we flag them for export. Lastly, we add a specialize directive to ensure that the new function `add` in module `DynSpec1` contains the desired instance for 7-element integer vectors.

When the above sample code is compiled, the vanilla SaC compiler already creates a new module with the desired instances. That new module can then be used as new provider of the `add` function instead of the original `Math` module.

In particular, the new module can be used as source for further specialisations of the function `add`. The same technique as in the above example can be applied where yet another module is created that imports the existing instances from the `DynSpec1` module created in the first round of specialisation. Using the new container, we can then add further additional instances.

All that remains to be done to exploit the existing machinery for runtime specialisation is to create the above code, at least in form of an abstract syntax tree in memory, start the compilation process and dynamically add the resulting library. This functionality, amongst other bookkeeping, is implemented in the specialisation controller.

In the simplest case, the controller dequeues a specialisation request, creates the corresponding abstract syntax tree to trigger the specialisation, enacts the compiler and collects back the updated library. That library is then dynamically linked to the program and the global registry is updated.

However, as the augmented program submits specialisation requests blindly, we might end up with many duplicate requests for specialisations that have already been performed. To prevent useless specialisation runs, the controller remembers requests it has acted upon and automatically disregards future requests of the same kind. Using this technique, we ensure that each request is only acted upon once. This guarantees that we at most one unnecessary specialisation attempt in cases where the requested instance had already been created statically.

As a further optimisation, to reduce the number of compiler runs, the controller can block multiple specialisations of the same function into a single abstract syntax tree. It suffices to include multiple `specialize` directives, one for each specialisation request.

Lastly, as the controller has a global overview over the requests submitted by the program over time, it can perform a form of frequency scheduling: Those specialisation requests that are enqueued particularly often can be acted upon first.

5 Related Work

A wealth of related work can be found in the area of runtime partial evaluation, often also referred-to as dynamic specialisation. Systems such as Tempo [9, 10], Fabius [11] or DyC [12] are based on user annotations which indicate to the compiler where dynamic specialisations can be expected. These systems then generate specific runtime specialisers leading to a staged compilation process. This measure keeps the overhead introduced by the compilation at runtime low. In contrast, our approach is based on the idea to specialise programs concurrently and asynchronously. This allows us to apply the full-fledged compiler to an annotated source code.

Further related work concerns approaches that operate on the code that is being executed. They typically analyse different instruction paths at runtime.

When it turns out that a certain path is used frequently, these paths are opimised further.

Dynamo [13] and DynamoRio [14] both identify hot spots in programs. When a hotspot has been identified execution is paused and optimised code is generated for it. As interpreting is expensive Dynamo tries to store as many optimised traces as possible in a trace cache. The next time a trace is executed dynamo points it to the optimised code stored in its cache.

Another approach, ADORE (Adaptive Object code RE-optimisation) [15], uses hardware performance monitoring to identify performance bottlenecks. Similar to the approach presented in this paper, ADORE uses two threads: One thread runs the application as it would have normally and the second thread runs the optimisation functions. However, the optimisations performed in the ADORE system primarily target insertions of data cache prefetching to improve the cache behaviour in subsequent runs.

6 Conclusion

We have presented an adaptive compilation framework for generic 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.

We are currently in the process of implementing the proposed compilation framework in the SaC compiler `sac2c`. It is still too early at this stage to quantify the benefits of the approach, and we postpone any detailed evaluation to future work.

It does not take much to identify a wealth of research questions arising from realising the proposed adaptive compilation framework. For example, given a number of available cores, what is a profitable division of cores into one group of cores that collaboratively execute the program and another group of cores that run dynamic specialisation controllers.

Furthermore, it would be more than reasonable to complete more than one specialisation request at a time, but rather take all such requests from the request queue that have been filed since the previous specialisation round. Although the dynamic invocation of the SaC-compiler is not on the ciritical path due to running concurrently with the main program on different computing resources, compiler runtimes are not completely irrelevant either. Therefore, it may be useful to run the SaC-compiler with a different option set in these cases.

Last not least, dynamic specialisation only makes sense for functions that actually benefit from the availability of more detailed structural information on argument arrays. This definitely holds for computationally intensive functions, but not so much, for example, for I/O-related functions. Identifying suitable functions alone is an interesting future research question.

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. Kreye, D.: A Compilation Scheme for a Hierarchy of Array Types. In Arts, T., Mohnen, M., eds.: Implementation of Functional Languages, 13th International Workshop (IFL'01), Stockholm, Sweden, Selected Papers. Volume 2312 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2002) 18–35
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. Marcussen-Wulff, N., Scholz, S.B.: On Interfacing SAC Modules with C Programs. In Mohnen, M., Koopman, P., eds.: 12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany. Volume AIB-00-7 of Aachener Informatik-Berichte., Technical University of Aachen (2000) 381–386
7. Grelck, C., Scholz, S.B., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
8. 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
9. Consel, C.: A general approach for run-time specialization and its application to c. In: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, USA, ACM Press (1996) 145–156
10. Noel, F., Hornof, L., Consel, C., Lawall, J.L.: Automatic, template-based run-time specialization: Implementation and experimental study. In: International Conference on Computer Languages, IEEE Computer Society Press (1998) 132–142
11. Leone, M., Lee, P.: Dynamic specialization in the fabius system. ACM Computing Surveys **30** (1998)
12. Grant, B., Philipose, M., Mock, M., Chambers, C., Eggers, S.J.: An evaluation of staged run-time optimizations in dyc. ACM SIGPLAN Notices **34** (1999) 293–304
13. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. SIGPLAN Not. **35** (2000) 1–12
14. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. International Symposium on Code Generation and Optimization (2003)
15. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and implementation of a lightweight dynamic optimization system. Journal of Instruction-Level Parallelism **6** (2004) 1–24