FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Implementation of a first SaC to μTC compiler

Deliverable D4.2, Issue 1.0

Workpackage WP4

Author(s):	Sven-Bodo Scholz, Stephan Herhut, Carl Joslin		
Reviewer(s):	Chris Jesshope		
WP/Task No.:	WP4	**Number of pages:**	44
Issue date:	31.1.09	**Dissemination level:**	Public

Purpose: The purpose of this deliverable is to give an overview on the status of the implementation of a the auto-parallelising SaC to μTC compiler and to discuss the challenges encountered.
Results: The main results of this deliverable are a first implementation of a SaC to μTC compiler, the documentation of the development process and a description of planned future extensions.
Conclusion: The main conclusions are as follows: We have devised a strategy to extend our research compiler by a new μTC back-end. This involves the design and implementation of a new optimisation technique, a lowering phase from SaC WITH-loops to μTC `create` operations and the extension of the memory model of our research compiler. Furthermore, we have designed and implemented a prototypical resource management solution. Lastly, we have identified a viable roadmap for further extensions to enhance the code generation and their implementation.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 28.5.09

Document history

When	Who	Comments
14.1.09	Stephan Herhut	Initial version
17.1.09	Sven-Bodo Scholz	Minor modifications
19.1.09	Sven-Bodo Scholz	Added section on implementation status
20.1.09	Stephan Herhut	Added appendices and an outlook on future work

SEVENTH FRAMEWORK PROGRAMME

Project co-funded by the European Commission within the 7^{th} Framework Programme (2007-11).

Table of Contents

1 Overview **1**

2 Structure of the Compiler **1**

2.1 Original Compilation Process . . . 2

2.2 Extended Compilation Process . . . 3

3 WITH-loop Flattening **4**

4 WITH-loop Slicing **5**

5 Memory Management **8**

6 Managing Resources **8**

7 Implementation Status **9**

8 Ongoing Work **9**

Appendices **10**

A Obtaining and Installing `sac2c` **11**

A.1 Example: TVD Solver for 2D Shock-Tube Problem . . . 11

B `sac2c` Manual Page **25**

1 Overview

The Apple-CORE project aims at developing many-core chip multi-processors, which we refer to as Microgrids, and a corresponding tool-chain consisting of an operating system layer and compilers for a low-level systems programming-language μTC [6], for the legacy language C with support for auto-parallelization and for the novel high-level data-parallel functional language SaC (Single Assignment C) [9]. We report in this document on the progress made on implementing a first compiler from SaC to the systems language of the Microgrid architecture μTC and discuss the challenges we have met.

In earlier publications [5, 4], we have analysed the Microgrid architecture in general and the μTC language in particular with respect to their suitability as a target for the SaC language. Our findings show that the Microgrid architecture with its support for fine-grained concurrency is an ideal match for the data-parallel programming paradigm of SaC. Furthermore, we have learned that μTC is a viable target language for SaC. However, to compile SaC to μTC, still a significant semantic gap needs to be bridged: Whereas the main data-parallel construct of SaC, *i.e.*, the WITH-loop, fully supports n-dimensional data-structures and data-parallel operations thereon, the corresponding operation of the systems language μTC, *i.e.*, the `create` construct for concurrent loops, is limited to one dimensional data-parallel operations.

To bridge this semantic gap, we have identified two solutions:

Flattening WITH-loops: Instead of performing the data-parallel operation on the high-level notion of an n-dimensional array, we map the element-wise operation directly onto the 1-dimensional data-vector, referred to as *ravel* in the following. This allows us to express an n-dimensional operation directly as a 1-dimensional `create`. However, if the computation of the single elements of the result requires the value of the abstract, n-dimensional index position, flattening the WITH-loop is not viable: The computations required to derive the n-dimensional index into the abstract array from the 1-dimensional offset into the concrete ravel would severely degrade if not offset the gains from the concurrent execution. For these cases, we have developed an alternative approach.

Nested `create` Operations: In case the abstract, n-dimensional index of a WITH-loop is required to compute the result of the WITH-loop, we map the WITH-loop to a nesting of `create` operations. As we have detailed in [5], for each dimension, we slice the result into a set of subarrays that is then computed concurrently using the `create` construct. As each dimension is represented by its own `create` operation, computing the n-dimensional index is comparatively cheap in this scenario: It suffices to concatenate the indices of the `create` operations. However, from an implementation perspective, this approach requires more effort. Support for slicing a result into partial results which are then computed independently had to be added to the compiler.

Before detailing these required extensions to our research compiler `sac2c`, we first give a coarse overview of the compiler's structure and the existing optimisations and identify where and how the above extensions are best introduced. Next, we describe the implementation of the WITH-loop flattening phase and our work on adding support for slicing of WITH-loops. Section 5 describes the required extensions to the memory subsystem of `sac2c`. A general discussion of resource management for the `create` instruction is given in Section 6. Finally, we summarise the status of the implementation and give an outlook on future work.

2 Structure of the Compiler

Our current research compiler `sac2c` has been developed over the course of more than 10 years. Before the start of the Apple-CORE project, the compiler supported C and C with POSIX threads as target languages. For the former, we are able to produce highly competitive sequential C code

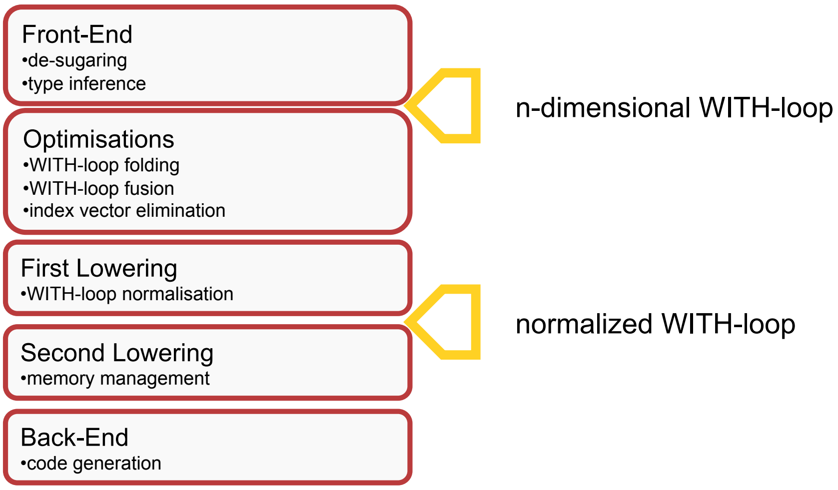

Figure 1: Overview of the main compilation stages during the translation of SaC programs to C using the `sac2c` compiler.

from high-level SaC specifications. This is achieved by applying more than 50 distinct optimisations during more than 200 compiler phases. By means of proprietary auto-paralellization techniques for the main data-parallel construct of SaC, the WITH-loop, we are furthermore able to produce efficient code for symmetric multi-processors using POSIX threads.

2.1 Original Compilation Process

Figure 1 gives an abstract overview of the compilation process. Due to the complexity of the compilation process and the number of optimisations involved, we can only give a very course overview here. We have only listed the most important steps in compilation, in particular those that are of importance for the implementation of the μTC support described in this report. A detailed description of all phases and the different intermediate languages used during compilation would be beyond the scope of this report.

As can be seen, the compilation process can be split into five stages. The first stage, the *front-end*, performs basic pre-processing steps to transform a SaC program into an equivalent de-sugared program in the language core of SaC. Furthermore, the program is annotated with type information. This information is used, apart for checking program correctness, at later stages to optimize the code.

The next stage is the *optimisation* stage. During this stage, all high-level optimisations are performed. High-level in this context refers to optimisations that can be performed on the SaC level, *i.e.*, those optimisations that can be implemented as source-to-source transformations. The most noteworthy optimisations in this context are WITH-LOOP FOLDING [8] and WITH-LOOP FUSION [2], which enhance the granularity of data-parallel operations by merging adjacent WITH-loops. A further optimisation that is performed during this stage is INDEX-VECTOR ELIMINATION [1], which translates, where possible, expressions that contain a reference to the index vector of a WITH-loop into equivalent expressions that use the offset into the ravel of the result instead.

During the third stage of the compiler, the *first lowering*, the high-level WITH-loop representation is lowered into a normalised form which, where possible, computes the result in canonical order. Introducing an explicit ordering of computation to the conceptually data-parallel WITH-loop allows us to perform enhanced optimisations such as cache blocking. However, the WITH-loop after this stage still remains n-dimensional.

The penultimate stage of the compilation process, the *second lowering*, introduces the notion of memory. Until this stage, SaC programs only use the notion of values and storage into memory is implicit. During this stage, the program is transformed in multiple steps into a program with

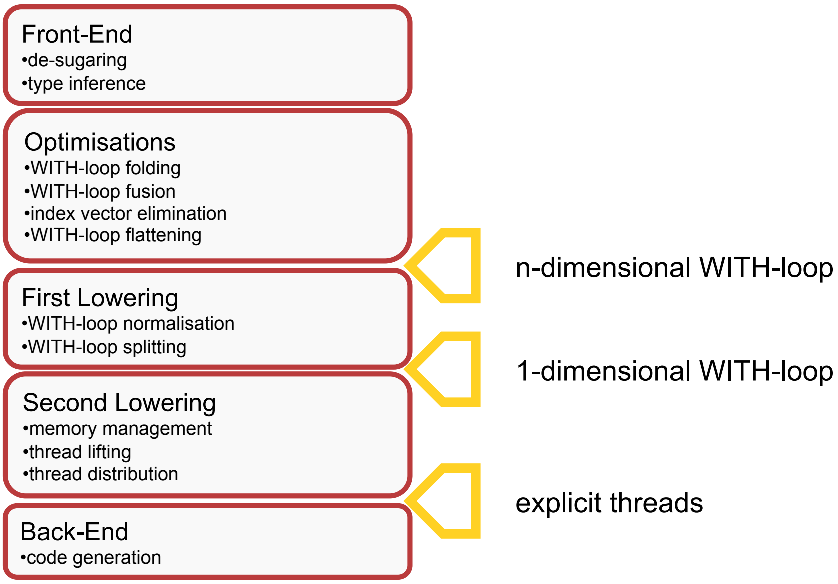

Figure 2: Overview of the extended compilation stages during the translation of SaC programs to C using the `sac2c` compiler.

explicit memory allocation and reference counting instructions. This stage is the first stage that is dependent on the compilation target. The memory allocation strategy differs for sequential and concurrent execution using the C and C with POSIX threads back-ends.

Finally, the last phase of compilation is the *back-end*. Depending on the target of compilation, a different back-end is used. Although both back-ends share a common infrastructure, the code generation, in particular for WITH-loops, is different.

2.2 Extended Compilation Process

To support μTC as a new target-language, three main extensions were required:

1. translation of n-dimensional WITH-loops into 1-dimensional `create` operations, where possible,
2. translation of n-dimensional WITH-loops into nested `create` operations, and
3. general support for producing μTC code in the back-end.

A key observation that allowed us to reduce the implementation effort is that the first extension above can be reduced to a special case of the second extension by mapping n-dimensional WITH-loops to 1-dimensional WITH-loops during the high-level optimisation stage. A 1-dimensional WITH-loop then automatically triggers the production of a non-nested `create` operation during the general translation of WITH-loops to `create` operations as outlined in [5]. This observation lead to the implementation of a new optimisation phase WITH-LOOP FLATTENING during the second stage of compilation. Figure 2 gives an overview of the extended compilation process. The new WITH-LOOP FLATTENING phase is performed directly after INDEX-VECTOR ELIMINATION. The latter, as it turns out, enables the flattening of WITH-loops even for some cases where the index vector is referenced in the WITH-loop body.

The second extension, the transformation of n-dimensional WITH-loops into nested `create` operations, is performed in two steps. We first transform the n-dimensional WITH-loop into a nesting of a new, one-dimensional WITH-loop representation. In a second step, this still relatively high-level representation is then lowered to the final nesting of `create` operations.

This two-step lowering is motivated by the requirement to lower the WITH-loop to its one-dimensional form before memory management. To introduce the notion of memory, we need to know how the computation will be sliced into sub-computations along the dimensions and what memory will be shared between threads and which memory is thread local. However, performing memory management on the loosely coupled `create` representation would inhibit many optimisations that make use of special properties of the WITH-loop.

The resulting compilation process can be seen in Figure 2. The third compilation stage, the first lowering, has been extended by a new With-Loop Splitting phase, which transforms n-dimensional WITH-loops into a nesting of a new one-dimensional WITH-loop construct. Furthermore, the second lowering stage has been extended to support memory management for this new one-dimensional WITH-loop. Once the memory management is complete, we then lower the representation further towards μTC by introducing the notion of threads to the intermediate representation. Nested one-dimensional WITH-loops are transformed into a nesting of threads during the THREAD LIFTING phase. Finally, the phase THREAD DISTRIBUTION performs some resource management.

The last required extension is to add support for emitting μTC code to the back-end. To keep the implementation effort for the new back-end for the μTC target language manageable, we have chosen to extend the existing C back-end. This decision was motivated by the fact that μTC is a superset of C and therefore most of the code-generation is expected to be similar. Only for data-parallel operations, *i.e.*, the WITH-loop construct of SaC, the code generation needs to be adapted to make use of the specific extensions of μTC for concurrent execution.

3 WITH-loop Flattening

As a first step, we have implemented the new With-loop Flattening optimisation. In general, With-loop Flattening is a source-to-source transformation on WITH-loops. A n-dimensional WITH-loop can be transformed into a semantically equivalent one-dimensional WITH-loop if it fulfils the following conditions:

1. the WITH-loop index is not referenced within the body of the WITH-loop, and
2. the WITH-loop comprises only a single full partition,

With *full* partition, we refer to a partition that computes the entire iteration space of the whole WITH-loop. For instance, for a `genarray` WITH-loop that computes a 4×4 matrix, a partition would be considered a full partition if it iterates all elements in the iteration space $[(0,0),(4,4))$.

The first condition can easily be checked by inspecting the set of free variables of the WITH-loop body. However, the second condition is more difficult to decide in general. It is, of course, straightforward to decide whether a WITH-loop comprises only a single partition. Whether such partition is a full partition is not decidable in general.

As we have no means to detect full partitions in general, we limit the applicability of With-loop Flattening in our current implementation to a subset of the theoretically transformable WITH-loops for which we can decide the second property above. As an approximation for whether a partition is a full partition, we use the following condition: For `modarray` and `genarray` WITH-loops, we flag a partition as full if

- the lower bound is the constant vector of zeros,
- the upper bound equates to the shape of the result, and
- the step and width parameters are the constant vector of ones.

A full description of the transformation scheme for With-loop Flattening would be beyond the scope of this report. However, to give an idea we provide a simple example:

```
A = with {
      ([0,0] <= iv < shape) : expr;
    } : genarray( shape, 0);
```

The above `genarray` WITH-loop has only a single partition which fulfils our criterion for full partitions as described above. Given that the body of the WITH-loop *expr* does not contain references to the index variable `iv`, the above code can be transformed into the following semantically equivalent WITH-loop:

```
1 r = prod( shape);
  An = with {
3        ([0] <= [i] < [r]) : expr;
      } : genarray( [r], 0);
5 A = reshape( shape, An);
```

As can be seen above, the new WITH-loop defined in Line 2 now is one-dimensional (note the one-element index `[i]`). It iterates over the full ravel of the result. The length of this ravel is computed in Line 1 as the product of all elements of the shape vector `shape` of the array to be computed. However, the above WITH-loop now computes a one-dimensional vector of length `r` instead of a two-dimensional array. This is remedied in Line 5 by modifying the shape of the result of the new WITH-loop to the shape of the result as specified for the old WITH-loop. Note that this operation does not incur any significant runtime cost as it in the worst case updates a descriptor and in no case needs to modify the data as such.

At first glance it may seem that the optimisation as described above only applies to a very small set of WITH-loops in practice. However, due to existing optimisations, in particular the INDEX-VECTOR ELIMINATION, WITH-loops like the one above are rather common in real-world programs. As an example, all basic map operations on arrays, *e.g.*, element-wise addition and multiplication, fall in the above category.

Although we have designed and implemented this optimisation specifically to support μTC as a compilation target, the flattening of n-dimensional WITH-loops has proven beneficial in general. By reducing the dimensionality of the iteration space of a WITH-loop, we are able to reduce the level of loop-nestings required to compute the result, as well. The resulting reduced overhead manifests in increased runtime performance.

We hope to publish a formal description of the WITH-LOOP FLATTENING transformation and quantitative results on the resulting runtime improvements (via μTC as well as via standard C) as soon as our toolchain is completed.

4 WITH-loop Slicing

The second extension we have implemented is the WITH-LOOP SLICING transformation performed during the first-lowering stage. In this phase the normalized, n-dimensional WITH-loop encoding used in the intermediate representation after the WITH-LOOP NORMALISATION phase is transformed into a new one-dimensional WITH-loop representation. This new WITH-loop representation was designed explicitly for a later mapping to the `create` construct of μTC. Apart from being one-dimensional only, the new representation differs in the following key aspects from the original n-dimensional version:

1. the WITH-loop index is no longer part of the WITH-loop but it is computed explicitly, and
2. the n-element *step* and *width* parameters are replaced by a single scalar *step* parameter.

A translation from the n-dimensional WITH-loop into the new one-dimensional WITH-loop needs to account for these differences.

The first difference, the explicit encoding of index computations, might seem like an arbitrary choice. However, it is a key requirement to be able to transform arbitrary n-dimensional WITH-loops into nestings of one-dimensional WITH-loops. To motivate this requirement, consider the following example:

```
A = with {
      ([0,0,0] <= iv < [4,3,4]) : B[iv];
    } : genarray( [4,3,4], 0);
```

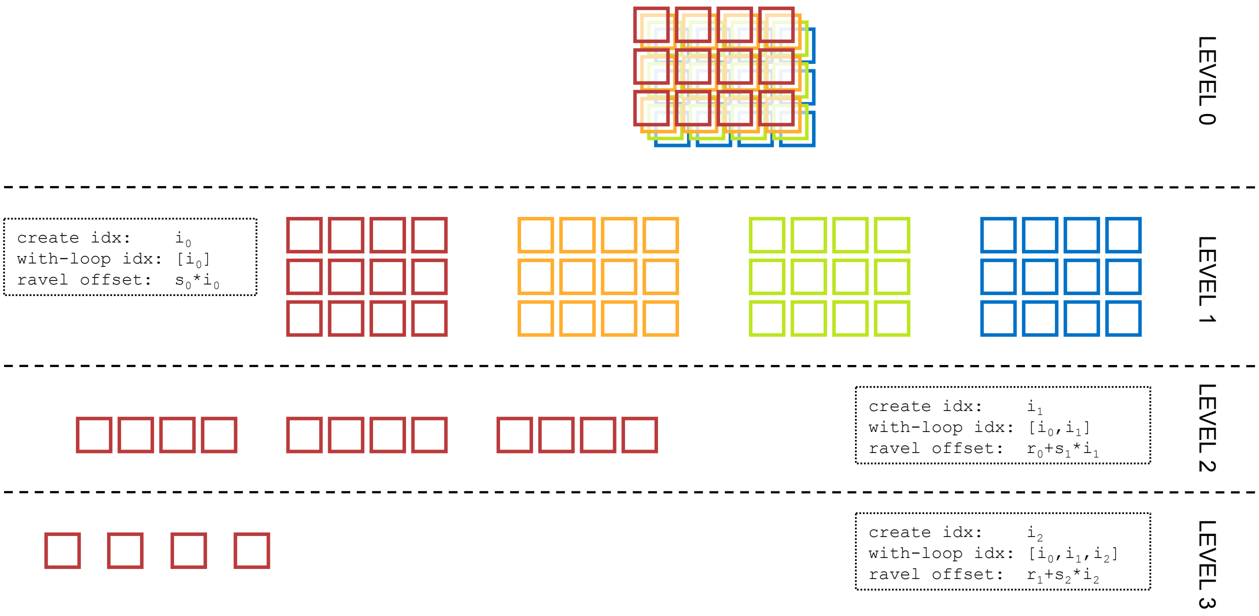

Figure 3: Graphical representation of the decomposition of an n-dimensional WITH-loop iteration space into one-dimensional loops and the corresponding index computations.

The above code copies a $4 \times 3 \times 4$ array `B` element-wise to a new array `A`. A schematic decomposition into one-dimensional loops is shown in Figure 3. At each level, the iteration space or sub-result computed by the current one-dimensional loop is shown. On the outermost level, the corresponding loop computes the entire result by slicing the result into 4 sub-results along the first dimension (depicted here as the z-axis). These sub-results are then computed by four loops on the first nesting level. Again, each loop slices the iteration space to be computed into sub-spaces, this time along the second dimension (depicted as the y-axis). For space reasons, Figure 3 shows the result of this slicing for the left-most loop only. As can be seen, the slicing yields three 4-element vectors as new sub-results to be computed on the second nesting level. Lastly, these are sliced into four scalar cells which can then be computed by single threads.

To compute the value of a scalar cell in the above example, two values are required for each thread at the leaves of the decomposition tree: The offset into the ravel of the result where the value needs to be written to and the 3-element index of the original WITH-loop to perform the selection into the source array `B`. However, the loop at each level considered in isolation only encodes the offset into the outer-most dimension of the current sub-result. To make the offset and index available to the threads, these need to be computed explicitly.

To reduce the computational complexity of offset and index computations, we use an encoding the pre-computes a partial offset and index at each level. For the simple case of a single partition with a step and width of 1 and scalar values at the inner-most nesting-level, the resulting computations for each level are shown in the dotted boxes in Figure 3.

In case of the WITH-loop index, we simply combine the indices of the nested one-dimensional loops to a vector. Note here that for more complex grid patterns, a single loop may not represent an entire dimension. In this case, computing the WITH-loop index is more complex: All indices of the one-dimensional loops that correspond to a single dimension need to be added.

For the offset into the ravel, we compute at each level the offset of the first element of the current sub-result by adding the current offset from the top-left corner of the sub-result of the previous level to the offset into the ravel computed at the previous level. This offset into the sub-result of the previous level is computed by multiplying the index of the `create` operation at the current level by the size of the sub-result one level below. The challenge here was to find an encoding that allows us to compute this size in the general case, *i.e.*, when the shape of the element is not known statically.

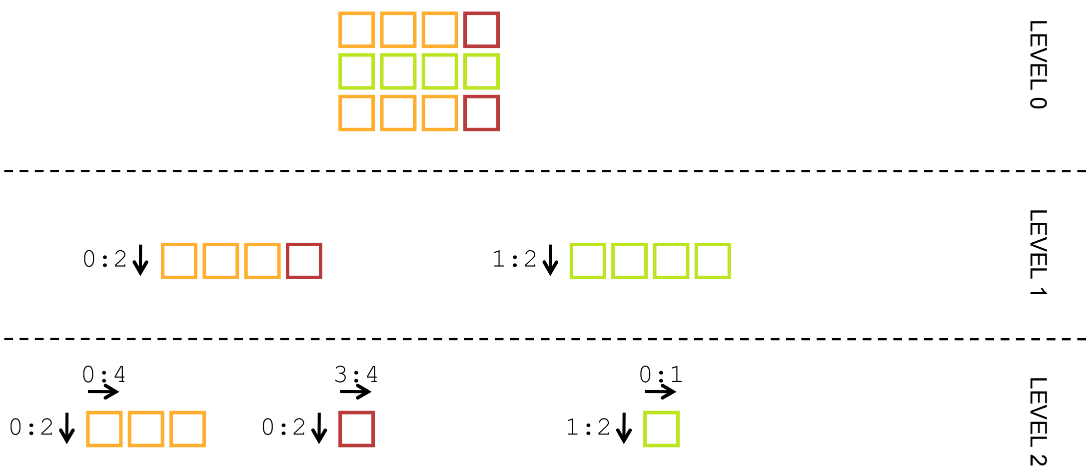

Figure 4: Graphical representation of the decomposition of a two-dimensional WITH-loop with `width` parameters into subcomponents that use only the `step` parameter.

This transformation only caters for the first difference between the n-dimensional WITH-loop and the one-dimensional encoding used to model `create` operations. However, we furthermore need to handle the second difference, *i.e.*, we need to translate WITH-loops that make use of the `width` parameter into semantically equivalent WITH-loops with a trivial width of one. As an example for a WITH-loop using both the `width` and `step` parameters, consider the following WITH-loop:

```
A = with {
      ([0,0]  <=  iv  <  [3,4]  step  [2,4]  width  [1,3])  :  expr1;
      ([0,3]  <=  iv  <  [3,4]  step  [2,4])  :  expr2;
      ([1,0]  <=  iv  <  [3,4]  step  [2,1])  :  expr3;
    } : genarray( [3,4], 0);
```

The first partition above makes use of a width parameter and therefore cannot be directly expressed as a μTC `create` operation. Instead, we first have to translate the above WITH-loop into a representation that only makes use of the step parameter. To achieve this, we first identify each unique component of the pattern described by the `step` and `width` parameters. Then, we express each non-scalar component by a `create` operation of its own. The resulting new pattern then no longer requires a `width` parameter.

To demonstrate this technique, we have depicted the pattern resulting from the above WITH-loop in the top third of Figure 4. The first partition computes the 3 element blocks starting at the top-left corner and repeating every 2 rows and 4 columns. The second partition fills the missing fourth element in the pattern of the first partition. This single element is repeated every 2 rows and 4 columns, as well. However, it starts with an offset of 3 columns. Finally, the last partition computes every second row starting with row two. As it computes the entire row, it has a stepping of 1 along the y-axis.

To resolve the width parameter and compute the step and offset of the repeating elements of the computed array, we decompose the pattern along each dimension into its components. For the above two-dimensional example, we thus need two decomposition steps. However, the approach scales to arbitrary numbers of dimensions as required by the WITH-loop in its most general form.

The result of the first decomposition is presented in the middle section of Figure 4. As can be seen, the pattern shown in the top section consists of two row-patterns. The first, shown on the left, computes every second row beginning with the first row. This is represent by the offset:step pair `0:2` in Figure 4. All other rows, *i.e.*, every second row starting with row two, are computed by the pattern given on the right side of Figure 4. The corresponding offset:step annotation is `1:2`.

Next, we decompose these row patterns along the remaining dimension. This yields the final three components of the pattern as shown in the bottom third of Figure 4. The first row-pattern is split into two components. The first component repeats every four elements and starts with the first element in each row. We have annotated this using the offset:step pair `0:4`. For the second component, which computes the remaining elements for this row-pattern, we similarly get a offset:step pair of `3:4`, *i.e.*, the pattern repeats every four elements and starts at the third element of the row.

The second row-pattern does not need to be split any further as it consists of a single component. Thus, we get an offset:step annotation for the second row-pattern in this dimension of `0:1`.

Using this decomposition into components, we can now apply the slicing technique described earlier for the simpler WITH-loop. However, instead of slicing the WITH-loop until we reach the computation of the inner-most elements, we now slice up to component level instead.

A full description of all transformations required to decompose the iteration spaces of WITH-loops in general into their components would be beyond the scope of this report. We refer the interested reader to our earlier publications on this technique in the context of WITH-LOOP NORMALISATION [3].

5 Memory Management

Once all n-dimensional WITH-loops have been transformed into nestings of one-dimensional WITH-loops, the next stage of the compilation process, the second-lowering stage, introduces explicit allocation and reference counting instructions.

To support the new one-dimensional WITH-loop, we had to extend the abstract memory model that underlies the memory management subsystem of the `sac2c` compiler in two aspects:

1. The notion of sub-result had to be introduced, and
2. support for allocating memory in a different context than it is used in had to be added.

The first amendment results from the slicing of n-dimensional WITH-loops into nestings of one-dimensional WITH-loops. Instead of one language construct to compute the result in a single step, this transformation produces multiple WITH-loops that each compute only a part of a single result array. This is nicely visualized for our example by Figure 3. On the inner-most level, only a single element of the array is computed and thus only the memory for that cell is required. One level further up, these single elements are then combined to an entire row. On level 1, these rows are then combined to two-dimensional results before, finally, these are concatenated to the result.

The change to the computation of arrays introduced by WITH-loop slicing invalidates an assumption that was previously built-into the memory subsystem of `sac2c`. Before we started implementing the μTC back-end, the memory subsystem conceptually always allocated memory for the entire result of an expression, *e.g.*, for the result of a whole WITH-loop. However, with the partial computation of results, now the memory for a previously single result might be allocated at multiple sites and only partially. Unfortunately, the existing model to describe the shape and dimensionality of allocated objects was not expressive enough to capture these changes. We have extended the memory model and its intermediate representation accordingly.

Secondly, in the existing memory model of the `sac2c` compiler, all memory was allocated in the same context in which it was initialised. With the introduction of threads, this assumption no longer holds. Consider again the result of the slicing in Figure 3. At the lowest level, each thread fills one cell of the 4-element vector allocated at the level above, which itself is a thread again. Thus, the memory is allocated in a different context than where it is first used. We have extended the memory model accordingly and taken first steps to allow for more explicit memory distribution between threads in the future.

6 Managing Resources

Our experiments on the impact of thread distribution on runtime performance published in [5] have shown that the implementation of the Microgrid architecture in the MGSim emulator is vulnerable to resource deadlocks if a too naïve thread distribution scheme is used.

We have identified two common reasons for resource deadlocks:

1. flooding the thread table with threads on intermediate levels of the concurrency tree and thus inhibiting the creation of the actual worker threads at the leaves, and
2. exhausting the maximum number of families due to a too deep nesting of `create` statements.

To prevent the first kind of resource deadlock, we have implemented an initial prototype of a throttling mechanism to ensure that sufficient threads remain at the leaves of the concurrency tree. We employ a program-global analysis that infers the maximum nesting level of WITH-loops in all reachable execution paths of a program. From this we then derive the maximum nesting level of `create` operations at runtime. Furthermore, for each WITH-loop level after slicing, the number of partitions is counted. This information is used to compute the width of the concurrency tree.

From this coarse model of the program-global concurrency tree, a distribution of the available maximum number of threads to one-dimensional WITH-loops is computed and annotated in the intermediate representation. These annotations are then used in the back-end to emit corresponding resource limits for the `create` statements.

Currently this distribution is static and the maximum number of threads available has to be passed to the `sac2c` compiler using the `maxthreads` compile time option. However, to achieve portability of binaries between different Microgrid implementations, it would be desirable to configure this parameter at runtime. Currently, different approaches, ranging from a runtime parameter for the executables to a special system-call to retrieve the parameters of the platform, are discussed.

For the second kind of resource deadlock we have not implemented a solution yet, as it is not clear whether this kind of resource deadlock should be handled by the systems language μTC instead. A possible solution at the μTC level would be to revert to a sequential execution of `create` statements once no more families can be created. Alternatively, the computation could be diverted to a different place that still has families available. However, should family induced resource deadlocks not be handled by the μTC language, one approach to prevent these at the SAC level would be to emit sequential code instead of `create` statements after a certain nesting depth of WITH-loop slices.

The implementation of resource management is still in its very early stages as we so far are not able to experiment with the simulation platform as a corresponding μTC compiler is not available. Furthermore, a simulation using the `utc-ptl` [10] libraries is not possible in this case, as the `utc-ptl` implementation has different resource constraints and in particular is not vulnerable to deadlocks.

7 Implementation Status

We have implemented all the required extensions as described in the previous sections. A precompiled binary distribution for multiple architectures is available from the Apple-CORE website at `http://www.apple-core.info/resources/`. We have tested this version of the compiler with the `utc-ptl` software implementation of the SVP model. We have used a slightly patched version of the third release of `utc-ptl`. The required patch, alongside a helper script, is available from the Apple-CORE website, as well. A detailed description of how to install and use `sac2c` for use with `utc-ptl` can be found in Appendix A.

8 Ongoing Work

The current implementation of the SAC to μTC compiler is only an initial prototype. First experiments and analyses of the generated code have already revealed a range of potential optimisations.

Firstly, the decomposition of WITH-loops into one-dimensional WITH-loops may lead to suboptimal nestings of `create` operations. In particular, our current implementation often generates thread families with very few threads. We expect that using a sequential implementation in these case to save on resources for further family creations might be advantageous. However, we have decided to postpone further research into this direction until we can perform a more exhaustive study on the Microgrid emulator.

A second optimisation is the extension of the With-loop Flattening optimisation to a wider range of partition and generator combinations. This, however, requires more sophisticated array-access analyses. We hope to be able to extend `sac2c` accordingly in the near future.

The code generation for fold with-loops offers a further potential for optimisation. Our current compilation scheme, as detailed in [5], is based on a sequential synchronisation. For sufficiently complex fold operations, it might be advantageous to use a different synchronisation scheme instead. Again, we would like to empirically study the current implementation on the Microgrid emulator first before trying a different synchronisation strategy.

As already mentioned in Section 6, the current resource analysis and management is based on rather simple heuristics. We expect that a more sophisticated analysis would allow us to improve on this. Furthermore, an extension of μTC with more explicit resource managing mechanisms might be of help in this respect, as well.

We will further exploit all the above optimisation potentials as soon as we are able to use the emulation platform to obtain realistic runtime estimates. In the meantime, we concentrate on those optimisations of which we already know that they in general improve runtimes, *e.g.*, an extended version of the With-Loop Flattening optimisation.

APPENDIX A - Obtaining and Installing `sac2c`

We have made a special version of the `sac2c` distribution that contains a binary of the `sac2c` compiler with added support for the μTC back-end available online. Periodicly updated archives for a range of platforms can be downloaded from the resources section of the Apple-CORE website at `http://www.apple-core.info/resources/`.

To compile SaC programs using μTC as back-end language, furthermore the `utc-ptl` Microgrid implementation (release 3) is required. `utc-ptl` can be obtained from its maintainer Michiel van Tol (`mwvantol@science.uva.nl`).

As `utc-ptl` is based on C++ and as it uses the standard library of C++, as well, it cannot be directly used to compile μTC programs that make use of the C standard-library. In particular, the implementation of the function `malloc` for allocating memory on the heap differs in `utc-ptl` from the corresponding function in the C standard-library. To circumvent these incompatibilities, we have implemented an adapter script `mutcc` (short for μTC compiler) that rewrites the μTC produced by `sac2c` such that it is compatible with `utc-ptl`.

As a further functionality, our adapter script emulates a standard C-compiler command-line interface. This allows us to use the `utc-ptl` μTC to C++ translator as a direct replacement for any standard C compiler. The adapter script is available in the resources section of the Apple-CORE website at `http://www.apple-core.info/resources/`, as well.

We have slightly modified the stock `utc-ptl` distribution to allow for a better integration with the `mutcc` helper script. An according patch is available online at the same location as the `mutcc` script itself.

To make `sac2c` aware of the `mutcc` script, it suffices to place the script into a directory where it can be found by the system shell, *i.e.*, in a directory listed in the `PATH` environment variable on UNIX systems, and to set the environment variable `UTCPTLHOME` to the directory where the `utc-ptl` distribution is located. Lastly, the `sac2c` configuration file `.sac2crc` in the user's home directory needs to be amended by the following two lines:

```
target utcptl:
CC              :=  "mutcc"
```

Note that the file might not yet exist or might be empty.

Once the `mutcc` adapter script has been set up as described above, the examples that come with the `sac2c` compiler (and of course any other valid SaC program) may be compiled to μTC using the following command line

```
sac2c -B mutc -target utcptl -O3 example.sac -o example
```

where `example.sac` is the name of the source file to compile and `example` is the name of the resulting binary executable. The first argument `-B mutc` chooses the μTC back-end over the default C99 back-end. By adding the `-target utcptl` flag, we instruct the `sac2c` compiler to use the `mutcc` script as back-end compiler. A full description of the `sac2c` compiler options is given in Appendix B.

A.1 Example: TVD Solver for 2D Shock-Tube Problem

We have successfully compiled the TVD solver example [7] using the μTC back-end and were able to show that a sufficient amount of concurrency is produced to utilize the latency hiding and concurrency features of the Microgrid architecture. However, as the toolchain for the Microgrid emulator `MGSim` is not yet available, we were unable to quantify what impact this has on the actual runtime compared to a sequential version.

The TVD solver implementation that we have used for our experiments is part of the demo suite that comes with the `sac2c` compiler distribution. To give an idea of its nature, we reproduce the source code here, as well.

```
/******************************************************************************
 *
```

```
 *  TVD solver for 2D shock-tube problem
 *
 *  Alexey Kudriavtsev, 2008
 *
 ******************************************************************************/

import StdIO: all;
import Array: all;
import ArrayIO: all;
import Math: all;
import File: all;

#define NSAVE 10

#ifndef OUTFILE_TECPLOT
#define OUTFILE_TECPLOT "outputs/Tecplot2d.dat"
#endif

#ifndef OUTFILE_GRID
#define OUTFILE_GRID "outputs/grid2d.dat"
#endif

#ifndef OUTFILE_FLOW
#define OUTFILE_FLOW "outputs/flow2d.dat"
#endif

/*******************************************************************************
 *
 * problem-specific constants:
 */

#ifndef NX                /* Number of cells along X */
#ifdef CAJ
#define NX 2000
#define NX4 2004
#else
#define NX 400
#define NX4 404
#endif
#endif

#ifndef NY                  /* Number of cells along Y */
#ifdef CAJ
#define NY 2000
#define NY4 2004
#else
#define NY 400
#define NY4 404
#endif
#endif

#ifndef XL                   /* Size of domain along X */
#define XL 2d
#endif

#ifndef YL                   /* Size of domain along Y */
#define YL 2d
#endif

#ifndef GAM                   /* Ratio of specific heats */
#define GAM 1.4d
#endif

#ifndef NJET                  /* Number of points across nozzle exit */
#define NJET 200
```

```
#endif

/*******************************************************************************
 *
 * algorithm configuration:
 */

#ifndef IADV               /* time integration method */
#define IADV 3
#endif

#ifndef IMUSCL             /* MUSCL reconstruction method */
#define IMUSCL 1
#endif

#ifndef IAXIS              /* Switch of plane/axisymmetric flow */
#define IAXIS 0
#endif


/*******************************************************************************
 *
 * derived constants:
 */

#define DX (XL/tod(NX))   /* Spatial increment along X */
#define DY (YL/tod(NY))   /* Spatial increment along Y */

/*******************************************************************************
 *
 * fixed constants:
 */

#define CFL 0.95d         /* Courant-Friedrichs-Levy number */
#define MS 2.2d           /* Shock wave Mach number */


/*
 *  Maximum extension of 1D arrays
 */
#ifndef nmax
#define nmax 400
#define nmax4 404
#endif

/*
 *  Energy as function of primitive variables
 */
inline double energ (double r, double p,
                     double ux, double uy)
{
    return(p/(GAM-1d)+0.5d*r*(ux*ux+uy*uy));
}

/*
 *  Pressure as function of conservative variables
 */
inline double press (double mx, double my,
                     double e, double r)
{
    return((GAM-1d)*(e-0.5d*(mx*mx+my*my)/r));
}

/*
 *  MIN_MOD limiter
```

```
 */
inline double MIN_MOD (double a, double b)
{
    if (a*b < 0d)
      c = 0d;
    else{
      if (fabs(a) < fabs(b))
        c = a;
      else
        c = b;}

    return (c);
}

/*
 *  Primitive variables from conservative ones
 */
specialize double[+] poststep (double[NX,NY,7] q);
inline
double[+] poststep (double[+] q)
{
    q = with { ([0,0,4] <= iv <= [NX-1,NY-1,4])
             : press(q[iv-[0,0,4]],q[iv-[0,0,3]],
               q[iv-[0,0,2]],q[iv-[0,0,1]]);
                 ([0,0,5] <= iv <= [NX-1,NY-1,5])
             : q[iv-[0,0,5]]/q[iv-[0,0,2]];
                 ([0,0,6] <= iv <= [NX-1,NY-1,6])
             : q[iv-[0,0,5]]/q[iv-[0,0,3]];}
             : modarray(q);

    return(q);
}

/*
 *  Cell-centered grid
 */
double[NX], double[NY] init_grid ()
{
    x = with { ([0] <= [ix] <= [NX-1])
            : DX*(tod(ix)+0.5d);}
            : genarray([NX], 0d);

    y = with { ([0] <= [iy] <= [NY-1])
            : DY*(tod(iy)+0.5d);}
            : genarray([NY], 0d);

    return (x,y);
}

inline
void save_step(double[+] x, double [+] y, double[+] q)
{
  save_grid( x,y);
  save_flow( q);
}

/*
 *  Saves grid to file
 */
inline
void save_grid (double[NX] x, double[NY] y)
{
    File ff;

    iv,ff = fopen ( OUTFILE_GRID,"w");
```

```

    for (ix=0; ix <= NX-1; ix++)
      fprintf(ff, "%lf \n", x[ix]);

    for (iy=0; iy <= NY-1; iy++)
      fprintf(ff, "%lf \n", y[iy]);

    fclose (ff);
}

/*
 *  Initial flowfield
 */
inline
double[NX,NY,7] init_flow ()
{
    u0 = 0d;
    v0 = 0d;
    p0 = 1d;
    r0 = GAM;
    e0 = energ(r0,p0,u0,v0);
    ru0 = r0*u0;
    rv0 = r0*v0;

    q = genarray([NX,NY], [ru0,rv0,e0,r0,p0,u0,v0]);

    return (q);
}

/*
 *  Saves flowfield to file
 */
inline
void save_flow (double[+] q)
{
    File ff;

    iv,ff = fopen ( OUTFILE_FLOW,"w");

    for (ix=0; ix <= NX-1; ix++){
    for (iy=0; iy <= NY-1; iy++){
      fprintf(ff, "%1.19lf\n%1.19lf\n%1.19lf\n%1.19lf\n\n",
      q[ix,iy,0], q[ix,iy,1], q[ix,iy,2], q[ix,iy,3]);
    }
    }
    fclose (ff);
}

/*
 *  Calls different subroutines for
 *  reconstructing cell-face values
 *  from cell-centered ones
 */
inline
double[nmax4,4], double[nmax4,4] muscl (double[nmax4,7] qc, double sx,
                                         double sy, int n1, int n2)
{
    qpl = genarray([nmax+4], [0d,0d,0d,0d]);
    qpr = genarray([nmax+4], [0d,0d,0d,0d]);

    if (IMUSCL == 1)
      qpl,qpr = muscl1 (qc, n1, n2);
    else if (IMUSCL == 2)
      qpl,qpr = pmuscl2 (qc, n1, n2);
    else if (IMUSCL == -2)
```

```
      qpl,qpr = xmuscl2 (qc, sx, sy, n1, n2);
    else if (IMUSCL == 3)
      qpl,qpr = weno3 (qc, sx, sy, n1, n2);
    else
      printf (" Wrong value of IMUSCL! \n");

    return (qpl,qpr);
}

/*
 *  Calculates cell-face values using
 *  1st order piecewise constant
 *  reconstruction
 */
specialize double[+], double[+] muscl1 (double[NX,NY,7] qc, int n1, int n2);
inline
double[+], double[+] muscl1 (double[+] qc, int n1, int n2)
{
    qpl = genarray([nmax+4,4], 0d);
    qpr = genarray([nmax+4,4], 0d);

    qpl = with { ([n1,0] <= iv <= [n2,3])
            : qc[iv+[0,3]];}
            : modarray(qpl);
    qpr = with{ ([n1,0] <= iv <= [n2,3])
            : qc[iv+[0,3]];}
            : modarray(qpr);

    return (qpl,qpr);
}

/*
 *  Calculates cell-face values using
 *  2nd order MUSCL reconstruction of
 *  primitive variables
 */
specialize double[+], double[+] pmuscl2 (double[NX,NY,7] qc, int n1, int n2);
inline
double[+], double[+] pmuscl2 (double[+] qc, int n1, int n2)
{
    qpl = genarray([nmax+4,4], 0d);
    qpr = genarray([nmax+4,4], 0d);

    dq = genarray([nmax+3,4], 0d);

    dq = with{ ([n1-1,0] <= iv <= [n2,3])
            : qc[iv+[1,3]]-qc[iv+[0,3]];}
            : modarray(dq);

    for (i=n1; i <= n2; i++){

       dql = [dq[i-1,0],dq[i-1,1],dq[i-1,2],dq[i-1,3]];

       dqr = [dq[i,0],dq[i,1],dq[i,2],dq[i,3]];

       for (L=0; L <=3; L++){

         gq = MIN_MOD(dql[L],dqr[L]);
         qpl[i,L] = qc[i,L+3]-0.5d*gq;
         qpr[i,L] = qc[i,L+3]+0.5d*gq;
       }
       }

    return (qpl,qpr);
}
```

```

/*
 * Calculates cell-face values using
 * 2nd order MUSCL reconstruction of
 * characteristic variables
 */
specialize double[+], double[+] xmuscl2 (double[NX,NY,7] qc, double sx,
                              double sy, int n1, int n2);
inline
double[+], double[+] xmuscl2 (double[+] qc, double sx,
                              double sy, int n1, int n2)
{
    qpl = genarray([nmax+4,4], 0d);
    qpr = genarray([nmax+4,4], 0d);

    dq = genarray([nmax+3,4], 0d);

    dq = with{ ([n1-1,0] <= iv <= [n2,3])
            : qc[iv+[1,3]]-qc[iv+[0,3]];}
            : modarray(dq);

    wq = genarray([4],0d);

    for (i=n1; i <= n2; i++){
       r  = qc[i,3];
       c2 = GAM*qc[i,4]/r;
       c  = sqrt(c2);

       dunl =  sx*dq[i-1,2]+sy*dq[i-1,3];
       dutl = -sy*dq[i-1,2]+sx*dq[i-1,3];
       dunr =  sx*dq[i,2]+sy*dq[i,3];
       dutr = -sy*dq[i,2]+sx*dq[i,3];

       wql = [dq[i-1,1]-r*c*dunl,
              dq[i-1,0]-dq[i-1,1]/c2,
              dutl,
              dq[i-1,1]+r*c*dunl];

       wqr = [dq[i,1]-r*c*dunr,
              dq[i,0]-dq[i,1]/c2,
              dutr,
              dq[i,1]+r*c*dunr];

       wq = with { ([0] <= L <= [2])
              : MIN_MOD(wql[L],wqr[L]);}
              : modarray(wq);

       gun = 0.5d*(wq[3]-wq[0])/(r*c);
       gut = wq[2];
       gp =  0.5d*(wq[0]+wq[3]);
       gr = gp/c2+wq[1];
       gux = sx*gun-sy*gut;
       guy = sy*gun+sx*gut;

       gq = [gr,gp,gux,guy];

       for (L=0; L<=3; L++){
         qpl[i,L] = qc[i,L+3]-0.5d*gq[L];
         qpr[i,L] = qc[i,L+3]+0.5d*gq[L];}

       }

    return (qpl,qpr);
}

```

```
/*
 *  Calculates cell-face values using
 *  3rd order WENO reconstruction of
 *  characteristic variables
 */
specialize double[+], double[+] weno3 (double[NX,NY,7] qc, double sx,
                            double sy, int n1, int n2);
inline
double[+], double[+] weno3 (double[+] qc, double sx,
                            double sy, int n1, int n2)
{
    eps = [0.00000001d, 0.00000001d,
           0.00000001d, 0.00000001d];
    s13 = 1d/3d;
    s23 = 2d/3d;


    qpl = genarray([nmax+4,4], 0d);

    qpr = genarray([nmax+4,4], 0d);

    dq = with{ ([n1-1,0] <= iv <= [n2,3])
            : qc[iv+[1,3]]-qc[iv+[0,3]];}
            : genarray([nmax+3,4], 0d);

    for (i=n1; i <= n2; i++){
       r  = qc[i,3];
       c2 = GAM*qc[i,4]/r;
       c  = sqrt(c2);

       dunl =  sx*dq[i-1,2]+sy*dq[i-1,3];
       dutl = -sy*dq[i-1,2]+sx*dq[i-1,3];
       dunr =  sx*dq[i,2]+sy*dq[i,3];
       dutr = -sy*dq[i,2]+sx*dq[i,3];

       wql = [dq[i-1,1]-r*c*dunl,
              dq[i-1,0]-dq[i-1,1]/c2,
              dutl,
              dq[i-1,1]+r*c*dunl];

       wqr = [dq[i,1]-r*c*dunr,
              dq[i,0]-dq[i,1]/c2,
              dutr,
              dq[i,1]+r*c*dunr];

       sl = wql*wql+eps;
       sr = wqr*wqr+eps;

       sl = sl*sl;
       sr = sr*sr;

       al = s13/sl;
       ar = s23/sr;

       bl = s23/sl;
       br = s13/sr;

       gwl = (bl*wql+br*wqr)/(bl+br);
       gwr = (al*wql+ar*wqr)/(al+ar);

       gunl = 0.5d*(gwl[3]-gwl[0])/(r*c);
       gutl = gwl[2];
       gpl  =  0.5d*(gwl[0]+gwl[3]);
       grl  = gpl/c2+gwl[1];
       guxl = sx*gunl-sy*gutl;
```

```
       guyl = sy*gunl+sx*gutl;

       gql = [grl,gpl,guxl,guyl];

       gunr = 0.5d*(gwr[3]-gwr[0])/(r*c);
       gutr = gwr[2];
       gpr  =  0.5d*(gwr[0]+gwr[3]);
       grr  = gpr/c2+gwr[1];
       guxr = sx*gunr-sy*gutr;
       guyr = sy*gunr+sx*gutr;

       gqr = [grr,gpr,guxr,guyr];

       for (L=0; L <= 3; L++){
         qpl[i,L] = qc[i,L+3]-0.5d*gql[L];
         qpr[i,L] = qc[i,L+3]+0.5d*gqr[L];}
       }

    return (qpl,qpr);
}

/*
 *  Evaluates numerical flux using
 *  HLLE approximate Riemann solver
 */
specialize double[+] flux (double[NX,NY,7] qpl, double[NX,NY,7] qpr,
                double sx, double sy, int n);
inline
double[+] flux (double[+] qpl, double[+] qpr,
                double sx, double sy, int n)
{
    f = genarray([nmax+1,4], 0d);

    f = with {
         ([0] <= [i] <= [n]) {

         rl = qpr[i+1,0];
         pl = qpr[i+1,1];
         ul = qpr[i+1,2];
         vl = qpr[i+1,3];
         unl = sx*ul+sy*vl;
         el = energ(rl,pl,ul,vl);
         ml = rl*ul;
         nl = rl*vl;
         cl = sqrt(GAM*pl/rl);

         ql = [ml, nl, el, rl];
         fl = [unl*ml+sx*pl, unl*nl+sy*pl, unl*(el+pl), unl*rl];

         rr = qpl[i+2,0];
         pr = qpl[i+2,1];
         ur = qpl[i+2,2];
         vr = qpl[i+2,3];
         unr = sx*ur+sy*vr;
         er = energ(rr,pr,ur,vr);
         mr = rr*ur;
         nr = rr*vr;
         cr = sqrt(GAM*pr/rr);

         qr = [mr, nr, er, rr];
         fr = [unr*mr+sx*pr, unr*nr+sy*pr, unr*(er+pr), unr*rr];

         bl = min(unl-cl,unr-cr);
         br = max(unl+cl,unr+cr);
         bm = min(bl,0d);
```

```
         bp = max(br,0d);

         fc = (bp*fl-bm*fr+
               bp*bm*(qr-ql))/(bp-bm);
         } : fc;
         } : genarray([nmax+1], [0d,0d,0d,0d]);


    return(f);
}

/*
 *  Advances solution in time
 */
inline
double[+], double step_flow (double[+] q, double t,
                             double tk)
{
    while ((fabs(t-tk) > 0.000000001d) ){
       dt = getdt(q);
       dt = min(dt,tk-t);

       if (IADV == 1)
         q = rktvd1 (q, dt);
       else if (IADV == 2)
         q = rktvd2 (q, dt);
       else if (IADV == 3)
         q = rktvd3 (q, dt);
       else
         printf (" Wrong value of IADV! \n");

       t = t + dt;
       printf("t = %1.16e, dt = %1.16e \n",t,dt);}

    return (q,t);
}

/*
 *  Evaluates available time step
 */
inline
double getdt(double[+] q)
{
    evmax = 0d;

    for (ix=0; ix <= NX-1; ix++){
    for (iy=0; iy <= NY-1; iy++){
      c = sqrt(GAM*q[ix,iy,4]/q[ix,iy,3]);
        ux = q[ix,iy,5];
        uy = q[ix,iy,6];
        ev = (fabs(ux)+c)/DX+(fabs(uy)+c)/DY;
        evmax = max(ev,evmax);
    }
    }

    dt = CFL/evmax;

    return (dt);
}

/*
 *  Time integration with forward Euler method
 */
specialize double[+] rktvd1 (double[NX,NY,7] q, double dt);
inline
```

```
double[+] rktvd1 (double[+] q, double dt)
{
    rq = rhs(q);

    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : q[iv] + dt*rq[iv];}
          : modarray(q);

    q = poststep (q);

    return (q);
}

/*
 *  Time integration with 2nd order Runge-Kutta method
 */
specialize double[+] rktvd2 (double[NX,NY,7] q, double dt);
inline
double[+] rktvd2 (double[+] q, double dt)
{
    q0 = q;

    rq = rhs(q);
    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : q[iv] + dt*rq[iv];}
          : modarray(q);
    q = poststep (q);

    rq = rhs(q);
    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : 0.5d*(q0[iv]+q[iv] + dt*rq[iv]);}
          : modarray(q);
    q = poststep (q);

    return (q);
}

/*
 *  Time integration with 3rd order
 *  Runge-Kutta TVD method
 */
specialize double[+] rktvd3 (double[NX,NY,7] q, double dt);
inline
double[+] rktvd3 (double[+] q, double dt)
{
    q0 = q;

    rq = rhs(q);
    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : q[iv] + dt*rq[iv];}
          : modarray(q);
    q = poststep (q);

    rq = rhs(q);
    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : 0.25d*(3d*q0[iv]+q[iv] + dt*rq[iv]);}
          : modarray(q);
    q = poststep (q);

    rq = rhs(q);
    q = with {([0,0,0] <= iv <= [NX-1,NY-1,3])
          : (q0[iv]+2d*q[iv] + 2d*dt*rq[iv])/3d;}
          : modarray(q);
    q = poststep (q);

```

```
    return (q);
}

/*
 *  Evaluates right hand side
 */
specialize double[+] rhs (double[NX,NY,7] q);
inline
double[+] rhs (double[+] q)
{
    gm1 = GAM-1d;
    gp1 = GAM+1d;

    ps  = (2d*GAM*MS*MS-gm1)/gp1;
    rs  = GAM*gp1*MS*MS/(gm1*MS*MS+2d);
    us  = 2d*(MS-1d/MS)/gp1;
    es  = energ(rs,ps,us,0d);

    qlbc = genarray([2,NY],[rs*us,0d,es,rs,ps,us,0d]);
    qlbc = with {
            ([0,NJET,0] <= iv=[ix,iy,L] <= [1,NY-1,6]){
              if ( (L == 0) || (L == 5) )
                qval = -q[3-ix,iy,L];
              else
                qval = q[3-ix,iy,L];
            } : qval;
           } : modarray(qlbc);

    qbbc = genarray([NX,2],[0d,rs*us,es,rs,ps,0d,us]);
    qbbc = with {
             ([NJET,0,0] <= iv=[ix,iy,L] <= [NX-1,1,6]){
               if ( (L == 1) || (L == 6) )
                 qval = -q[ix,3-iy,L];
               else
                 qval = q[ix,3-iy,L];
             } : qval;
             } : modarray(qbbc);

    rq = genarray([NX,NY,4],0d);

    qc = genarray([nmax+4,7],0d);

    f = genarray ([nmax+1],[0d,0d,0d,0d]);

    sx = 1d; sy = 0d;

    for (iy=0; iy <= NY-1; iy++){

      qc = with {([2,0] <= iv=[ix,L] <= [NX+1,6])
             : q[ix-2,iy,L];}
             : modarray(qc);

      qc = with {([0,0] <= iv=[ix,L] <= [1,6])
             : qlbc[ix,iy,L];}
             : modarray(qc);

      qc = with {([NX+2,0] <= iv=[ix,L] <= [NX+3,6])
             : qc[NX+1,L];}
             : modarray(qc);

      qpl,qpr = muscl (qc, sx, sy, 1, NX+2);

      f = flux(qpl,qpr, sx, sy, NX);
      rq = with {([0,iy,0] <= iv=[ix,j,L] <= [NX-1,iy,3])
             : rq[iv]+(f[ix,L]-f[ix+1,L])/DX;}
```

```
             : modarray(rq);
    }

    qc = genarray([nmax+4,7],0d);

    f = genarray ([nmax+1],[0d,0d,0d,0d]);

    sx = 0d; sy = 1d;

    for (ix=0; ix <= NX-1; ix++){

      qc = with {([2,0] <= iv=[iy,L] <= [NY+1,6])
             : q[ix,iy-2,L];}
             : modarray(qc);

      qc = with {([0,0] <= iv=[iy,L] <= [1,6])
             : qbbc[ix,iy,L];}
             : modarray(qc);

      qc = with {([NY+2,0] <= iv=[iy,L] <= [NY+3,6])
             : qc[NY+1,L];}
             : modarray(qc);

      qpl,qpr = muscl (qc, sx, sy, 1, NY+2);

      f = flux(qpl,qpr, sx, sy, NY);

      rq = with {([ix,0,0] <= iv=[i,iy,L] <= [ix,NY-1,3])
             : rq[iv]+(f[iy,L]-f[iy+1,L])/DY;}
             : modarray(rq);
    }

     if (IAXIS == 1){
       y = with {([0] <= [iy] <= [NY-1])
             : DY*(tod(iy)+0.5d);}
             : genarray([NY], 0d);
       rq = with {
             ([0,0,0] <= iv=[ix,iy,L] <= [NX-1,NY-1,3])
           {qq = q[iv];
            if (L == 2){
              qq = qq + q[iv+[0,0,2]];}
            uy = q[ix,iy,6];
            yc = y[iy];
            rqval = rq[iv]-qq*uy/yc;} : rqval;
           } : modarray(rq);
    }

    return(rq);
}

/*
 *  Main program
 */
int main()
{
    tf = 0.05d;
    tp = 0.05d;

    x,y = init_grid();
    q = init_flow();

#if defined (SAVE)
    save_step( x,y,q);
#endif

```

```
    t = 0d;
    tk = t;
    while (t < tf) {
      tk = tk+tp;
      q,t = step_flow(q,t,tk);

#if defined (SAVE)
      printf ("\n record at t = %lf \n \n",t);
      save_step( x,y,q);
#endif
    }

    return(0);
}
```

APPENDIX B - `sac2c` Manual Page

```
---------------------------------------------------------------------------
 SAC - Single Assignment C
---------------------------------------------------------------------------


NAME:          sac2c
VERSION:       v1.00-beta (Buchette d'Anjou)
PLATFORM:      darwin9.7.0_i686


DESCRIPTION:

    The sac2c compiler transforms SAC source code into executable programs
    (SAC programs) or into a SAC specific library format (SAC module and
    class implementations), respectively.

    The compilation process is performed in 4 separate stages:
      1. sac2c uses any C preprocessor to preprocess the given SAC source;
      2. sac2c itself transforms preprocessed SAC source code into C code;
      3. sac2c uses any C compiler to generate target machine code;
      4. sac2c uses any C linker to create an executable program
         or sac2c itself creates a SAC library file.

    When compiling a SAC program, sac2c stores the corresponding
    intermediate C code either in the file a.out.c in the current directory
    (default) or in the file <file>.c if <file> is specified using the -o
    option. Here, any absolute or relative path name may be used.
    The executable program is either written to the file a.out or to any
    file specified using the -o option.

    However, when compiling a SAC module/class implementation, the
    resulting SAC library is stored in the files <mod/class name>.a
    and  <mod/class name>.so in the current directory.
    In this case, the -o option may be used to specify a
    different directory but not a different file name.


SPECIAL OPTIONS:

    -h              Display this helptext.
    -help           Display this helptext.
    -copyright      Display copyright/disclaimer.
    -V              Display version identification.
    -VV             Display verbose version identification.

    -libstat        Print status information of the given SAC library file.
    -prsc           Print resource settings.

    -M              Detect dependencies from imported modules/classes and
                    write them to stdout in a way suitable for the make
```

```
                    utility.
    -Mlib           Same as -M except that the output format is suitable
                    for makefiles used by the standard library building
                    process.

    NOTE:
    When called with one of these options, sac2c does not perform
    any compilation steps.


GENERAL OPTIONS:

    -D <var>        Set preprocessor variable <var>.
    -D <var>=<val>  Set preprocessor variable <var> to <val>.
    -cppI <path>    Specify path for preprocessor includes.

    -L <path>       Specify additional SAC library file path.
    -I <path>       Specify additional SAC library source file path.
    -E <path>       Specify additional C library file path.

    -o <name>       For compilation of programs:
                      Write executable to specified file.
                    For compilation of module/class implementations:
                      Write library to specified directory.

    -c              Generate C-file only; do not invoke C compiler.

    -v <n>          Specify verbose level:
                      0: error messages only
                      1: error messages and warnings
                      2: basic compile time information
                      3: full compile time information
                      4: even more compile time information
                    (default: 3)


BREAK OPTIONS:

    Break options allow you to stop the compilation process
    after a particular phase, subphase or cycle optimisation.
    By default the intermediate programm will be printed,
    but this behaviour may be influenced by the following
    compiler options:

    -noPAB          Deactivates printing after break.
    -doPAB          Activates printing after break.

    -b<spec>        Break after the compilation stage given
                    by <spec>, where <spec> follows the pattern
                    <phase>:<subphase>:<cyclephase>:<pass>.
                    The first three are from the list of
                    encodings below. The last one is a natural
                    number. Alternatively, a number can be used
```

```
                    for the phase, as well.


BREAK OPTION SPECIFIERS:

    scp | 1  : Loading SAC program
      loc      : Locating source code
      cpp      : Running C preprocessor
      prs      : Parsing input file

    pre | 2  : Preprocessing SAC program
      hs       : Hiding struct definitions behind typedefs and accessors
      iotc     : Introducing user-tracing calls
      zgwl     : Handling zero-generator with-loops
      mgwl     : Handling multi-generator with-loops
      mowl     : Handling multi-operator with-loops
      acn      : Resolving axis control and dot notation
      rpr      : Resolving pragma annotations
      obi      : Generating object initializers
      csgd     : Checking and simplifying generic definitions

    mod | 3  : Running module system
      rsa      : Processing use and import statements
      ans      : Annotating namespaces
      gdp      : Gathering dependencies
      pdp      : Printing dependencies
      imp      : Retrieving imported symbols
      uss      : Retrieving used symbols
      asf      : Loading prelude functions

    sim | 4  : Simplifying source code
      w2d      : Transforming while-loops into do-loops
      ece      : Eliminating conditional expressions
      moe      : Handling multiple operator expressions
      flt      : Flattening nested expressions
      udt      : Processing user defined types
      ggtc     : Generating generic type conversion functions

    ptc | 5  : Converting to static single assignment form
      ivd      : Inserting variable declarations
      itc      : Converting type decls into type conversions
      cwf      : Creating wrapper functions
      gon      : Running global object analysis
      goi      : Generating global object initialiser
      rso      : Resolving global objects
      rrp      : Resolving reference parameters
      ewt      : Extending dispatch information
      l2f      : Eliminating loops and conditionals
      elf      : Extending LaC funs
      ssa      : Establishing static single assignment form

    tc  | 6  : Running type inference system
      esp      : Enforcing Specializations
```

```
  sossk    : Specialization Oracle for Static Shape Knowledge
  ti       : Running type inference system
  etv      : Eliminating Type Variables
  ebt      : Eliminating Bottom Types
  swr      : Splitting Wrappers

exp | 7  : Processing exports
  exp      : Exporting symbols
  dfr      : Removing dead functions
  ser      : Serializing syntax tree
  rgd      : Removing generic function definitions
  iif      : Restoring bodies of imported inline functions

unq | 8  : Checking uniqueness property of objects
  cua      : Checking uniqueness annotations
  cuq      : Checking uniqueness

cwc | 9  : Creating Wrapper Code and Eliminating User-Defined Types
  cwb      : Creating Wrapper Bodies
  l2f      : Eliminating conditionals in wrapper code
  ssa      : Establishing static single assignment form in wrapper code
  dfc      : Trying to dispatch functions statically
  eudt     : Eliminating User-Defined Types
  icc      : Inserting Conformity Checks
  ti       : Running type inference system
  etv      : Eliminating Type Variables
  ebt      : Eliminating Bottom Types

ewl | 10 : Enhancing with-loops
  accu     : Introducing explicit accumulators
  adp      : Adding default partitions
  wlpg     : Generating full with-loop partitions

opt | 11 : Running SAC optimizations
  dfr      : Removing dead functions
  inl      : Applying function inlining
  dfr2     : Removing dead functions
  dcr      : Removing dead code
  lir      : Applying loop invariant removal
  isaa1    : Inserting symbolic array attributes
  esaa1    : Eliminating symbolic array attributes
  saadcr   : Removing dead code (after SAA cycle 1)
  glf      : Grouping local functions
  cyc      : Optimization cycle
    cse      : Applying common subexpression elimination (fun based)
    ili      : Inferring loop invariant variables (fun based)
    tup      : Applying type upgrade (fun based)
    etv      : Eliminating Type Variables (fun based)
    ebt      : Eliminating Bottom Types (fun based)
    dfc      : Applying function call dispatch (fun based)
    inl      : Applying inlining (fun based)
    wlpr     : Applying with-loop propagation (fun based)
    cf       : Applying constant folding (fun based)
```

```
    cvp      : Propagating constants and variables (fun based)
    wlpg     : Generating full with-loop partitions (fun based)
    wlsimp   : Simplifying with-loops (fun based)
    cwle     : Eliminate copy with-loops (fun based)
    wli      : Inferring foldable with-loops (fun based)
    wlf      : Applying with-loop folding (fun based)
    wlfssa   : Restoring SSA form after with-loop folding (fun based)
    shwlc    : Activating display of WL-Cost information (fun based)
    unshwlc  : Deactivating display of WL-Cost information (fun based)
    dcr      : Applying dead code removal (fun based)
    wls      : Applying with-loop scalarization (fun based)
    prfunr   : Applying prf unrolling (fun based)
    lur      : Applying loop unrolling (fun based)
    lurssa   : Restoring SSA form after loop unrolling (fun based)
    wlur     : Applying withloop unrolling (fun based)
    wlurssa  : Restoring SSA form after withloop unrolling (fun based)
    linl     : Inlining degenerated LaC functions (fun based)
    wlir     : Applying with-loop invariant removal (fun based)
    etc      : Eliminating typeconv primitives (fun based)
    esd      : Eliminating subtraction and division operators (fun based)
    as       : Arithmetic Simplification (fun based)
    al       : Applying associative law (fun based)
    dl       : Applying distributive law (fun based)
    uesd     : Reintroducing subtraction and division operators (fun based)
    dcr2     : Applying dead code removal (fun based)
    sisi     : Simplifying function signatures
    lof      : Lifting optimization flags
  scyc     : Type stabilization cycle
    tup      : Applying type upgrade (fun based)
    etv      : Eliminating Type Variables (fun based)
    ebt      : Eliminating Bottom Types (fun based)
    dfc      : Applying function call dispatch (fun based)
    lof      : Lifting optimization flags
  uglf     : Ungrouping local functions
  ls       : Applying Loop Scalarization
  lir2     : Applying loop invariant removal
  dfr3     : Removing dead functions
  flt      : Flattening with-loop generators
  ivext    : Inserting index vector extrema
  dcr2     : Applying dead code removal again
  isaa2    : Inserting symbolic array attributes
  saacyc   : Symbolic array attribute cycle 2
    prfunr   : Applying prf unrolling
    tup      : Applying type upgrade
    etv      : Eliminating type variables
    ebt      : Eliminating bottom types
    cf       : Applying constant folding
    cse      : Eliminating common subexpressions
    cvp      : Propagating constants and variables
    wlpg     : Generating full with-loop partitions
    wlsimp   : Simplifying with-loops
    ivexp    : Propagating index vector extrema
    swlfi    : Inferring symbolically foldable with-loops
```

```
    swlf     : Applying symbolic with-loop folding
    dcr      : Removing dead code
  tup      : Running final type inference
  etv      : Eliminating type variables
  ebt      : Eliminating bottom types
  wlfs     : Applying with-loop fusion
  wlfscse  : Eliminating common subexpressions after fusion
  wlfsdcr  : Removing dead code after fusion
  wlpg2    : Generating full with-loop partitions
  wrci     : Inferencing with-loop reuse candidates
  wlidx    : Annoting offset variable at with-loops
  ivexc    : Cleaning up index vector extrema
  scc      : Stripping conformity checks and dataflow guards
  ivesplit : Eliminating index vectors (split selections)
  ivecvp   : Propagating constants and variables (for IVE)
  ivecse   : Eliminating common subexpression (for IVE)
  iveras   : Eliminating index vectors (reuse WL-offsets and scalarize)
  wlflt    : Trying to flatten multi-dimensional withloops
  esaa2    : Eliminating symbolic array attributes
  lir3     : Applying loop invariant removal
  ufl      : Unflattening WL generator
  dcr3     : Removing dead code
  wllom    : Withloop lock optimization marking
  wllos    : Withloop lock optimization shifting
  fdi      : Freeing dispatch information
  pfap     : Profiling function applications
  stat     : Displaying optimisation statistics

wlt | 12 : Transforming with-loop representation
  ussa     : Converting from SSA form
  f2l      : Reintroducing loops and conditionals
  linl     : Inlining LaC functions
  wltr     : Transforming with-loop representation
  l2f      : Eliminating loops and conditionals
  ssa      : Establishing static single assignment form
  wlsd     : Splitting withloops by dimensions
  cvp      : Propagating constants and variables
  dcr      : Removing dead code
  acuwl    : Annotate CUDA withloops
  cutycv   : CUDA type conversion

mt3 | 13 : Running 3rd generation multithreading
  tem      : Tagging execution modes
  crwiw    : Creating with in with
  pem      : Propagating execution modes
  cdfg     : Creating data flow graph
  asmra    : Rearringing assignments
  crece    : Creating execution mode cells
  cegro    : Extending execution mode cells
  repfun   : Replicating functions
  mtdfr    : Removing superfluous functions
  concel   : Consolidating execution mode cells
  abort    : Aborting MT3 compilation
```

```
mem | 14 : Introducing memory management instructions
  simd     : SIMD inference
  asd      : AUD/SCL distinction
  copy     : Making copy operations explicit
  racc     : Removing alias results from conformity checks
  alloc    : Introducing explicit allocation statements
  dcr      : Removing dead code
  rci      : Inferring reuse candidates
  shal     : Activating display of alias information
  ia       : Interface aliasing analysis
  lro      : Applying loop reuse optimization
  aa       : Aliasing analysis
  srce     : Removing non-local reuse-candidates
  frc      : Removing invalid reuse candidates
  sr       : Static reuse
  rb       : Introducing reuse branches
  ipc      : Identifying in-place updates
  dr       : Exploiting data reuse
  dcr2     : Removing dead code again
  unshal   : Deactivating display of alias information
  rc       : Running reference count inference
  rcm      : Reducing reference counting instructions
  rco      : Optimizing reference counting instructions
  re       : Removing reuse instructions

ussa | 15 : Converting from static single assignment form
  ussa     : Converting from SSA form
  f2l      : Reintroducing loops and conditionals
  linl     : Inlining LaC functions
  rec      : Removing external code
  rera     : Restoring reference arguments
  reso     : Restoring global objects

mt  | 16 : Running automatic parallelisation
  mtcm     : Running multithreading cost model
  mtstf    : Creating MT and ST functions
  mtspmdf  : Creating SPMD functions
  mtas     : Annotating scheduling information
  sspmdls  : Applying SPMD linksign pragma

pc  | 17 : Preparing C code generation
  cuknl    : Create Cuda kernel functions
  lw3      : Lifting With-Loop bodies into threads
  mmv      : Marking memval identifiers
  dst      : Computing static thread mapping
  sls      : Applying linksign pragma
  moi      : Manage object initialisers
  rcs      : Resolving code sharing in With-Loops
  fpc      : Reorganising function prototypes
  tcp      : Applying type conversions
  mng      : Mark NoOp Grids
  rid      : Consistently renaming identifiers
```

```
    cg  | 18 : Generating Code
      tp       : Tag preparation
      ctr      : Converting to old type representation
      cpl      : Creating intermediate code macros
      prt      : Generating C file(s)
      frtr     : De-allocating syntax tree representation

    icc | 19 : Creating binary code
      hdep     : Handling dependencies
      ivcc     : Invoking C compiler
      crl      : Creating SAC library


PRINTING OPTIONS:

    -print [adv]+
               Add internal AST information as comments to the program output.
               The following flags are supported:
                 a: Print all (same as dv).
                 d: Print specialization demand.
                 v: Print avis information.


TYPE INFERENCE OPTIONS:

    -specmode <strat>  Specify function specialization strategy:
                         aks: try to infer all shapes statically,
                         akd: try to infer all ranks statically,
                         aud: do not specialize at all.
                         (default: aks)

    -maxspec <n>       Individual functions will be specialized at most <n> times.
                         (default: 20)


OPTIMIZATION OPTIONS:

    -enforceIEEE    Treat floating point arithmetic as defined in the IEEE-754
                    standard. In particular, this means
                      - disable some algebraic optimizations,
                      - disable segmentation and tiling of fold-with-loops,
                      - disable parallel execution of fold-with-loops.
                    Currently implemented for:
                      - associative law optimization,
                      - segmentation and tiling of fold-with-loops.

    -noreuse        Disable reuse inference in emm.

    -iveo <n>       Enable or disable certain index vector optimisations
                    <n> is a bitmask consisting of the following bits:
                      1: enable the usage of withloop offsets where possible
                      2: scalarise vect2offset operations where possible
```

```
                  3: try to optimise computations on index vectors
                  4: try to reuse offsets once computed
                The iveo option to for testing, and is to be removed.

-ssaiv          This option, if enabled, forces all with-loop generator
                variables to be unique (SSA form). (This is
                a prerequisite for MINVAL/MAXVAL work.)

                If disabled (the default setting), all with-loop
                generators use the same index vector variables.

-extrema        This option, if enabled, allows the compiler to
                use optimizations based on index variable extrema;
                i.e., the minimum and maximum value that index variables
                may take on. This option requires that -ssaiv is also enabled.

-glf            With this option local functions (loop, cond, ...) are
                grouped together in a local spine during the optimisation
                This is an internal option only.

-no <opt>       Disable optimization technique <opt>.

-do <opt>       Enable optimization technique <opt>.


The following optimization techniques are currently supported:

 (A leading * identifies optimization enabled by default.)

  * ls      loop scalarization
  * dcr     dead code removal
  * cf      constant folding
  * lir     loop invariant removal
  * inl     function inlining
  * lur     loop unrolling
  * wlur    with-loop unrolling
  * prfunr  prf unrolling
    lus     loop unswitching
  * cse     common subexpression elimination
  * dfr     dead function removal
    wlt     with-loop transformation
  * wlf     with-loop folding
    swlf    symbolic with-loop folding
  * ive     index vector elimination (requires -dosaa)
    wlflt   withloop flattening
    ae      array elimination
  * dl      distributive law
  * rco     reference count optimization
  * uip     update-in-place analysis
  * dr      data reuse
  * ipc     in-place computation
    tsi     with-loop tile size inference
    tsp     with-loop tile size pragmas
```

```
  * wlpg    with-loop partition generation
  * cvp     constant and variable propagation
  * srf     static reuse / static free
    phm     private heap management
    aps     arena preselection (requires -dophm)
    dpa     descriptor preallocation (requires -dophm)
    msca    memory size cache adjustment (requires -dophm)
    ap      array padding
    apl     array placement
  * wls     with-loop scalarization
  * al      associative law
  * as      arithmetic simplification
  * etc     typeconv elimination
    sp      selection propagation
  * wlsimp  with-loop simplification
  * cwle    copy with-loop elimination
    wlfs    with-loop fusion
  * lro     loop reuse optimization
  * tup     type upgrade
    sisi    signature simplification
  * sde     subtraction / division elimination
  * wlprop  with-loop propagation
  * saa     use symbolic array attributes
  * cyc     run optimization cycle
  * scyc    run stabilization cycle
    wllo    run with-loop lock optimization

NOTE:
 -no opt     disables all optimizations at once.
 -do opt     enables all optimizations at once.

NOTE:
 Upper case letters may be used to indicate optimization techniques.

NOTE:
 Command line arguments are evaluated from left to right, i.e.,
 "-no opt -do inl" disables all optimizations except for function inlining.

NOTE:
 Some of the optimization techniques are parameterized by additional side
 conditions. They are controlled by the following options:

-maxoptcyc <n>  Repeat optimization cycle max <n> times. After <n> cycles
                all optimisations except for type upgrade and function dispatch
                are disabled.
                  (default: 10)

-maxrecinl <n>  Inline recursive function applications at most <n> times.
                  (default: 0)

-maxlur <n>     Unroll loops having at most <n> iterations.
                  (default: 2)
```

```
-maxwlur <n>    Unroll with-loops with at most <n> elements generator set
                size.
                  (default: 9)

-maxae <n>      Try to eliminate arrays with at most <n> elements.
                  (default: 4)

-initmheap <n>  At program startup initially request <n> KB of heap memory
                for master thread.
                  (default: 1024)

-initwheap <n>  At program startup initially request <n> KB of heap memory
                for each worker thread.
                  (default: 64)

-inituheap <n>  At program startup initially request <n> KB of heap memory
                for usage by all threads.
                  (default: 0)

-aplimit <n>    Set the array padding resource allocation overhead limit
                to <n> %.
                  (default: 10)

-apdiag         Print additional information for array padding to file
                "<outfile>.ap", where <outfile> is the name specified via
                the "-o" option.

-apdiagsize <n> Limit the amount of information written to the diagnostic
                output file created via the -apdiag option to approximately
                <n> lines.
                  (default: 20000)

-wls_aggressive Set WLS optimization level to aggressive.
                WARNING:
                Aggressive with-loop scalarization may have the opposite
                effect as with-loop invariant removal and cause duplication
                of code execution.

-maxwls         Set the maximum number of inner with-loop elements for which
                aggressive behaviour will be used even if -wls_aggressive is
                not given. (default: 1)

-nofoldfusion   Eliminate fusion of with-loops with fold operator.

-maxnewgens <n> Set the maximum number of new created generators while
                intersection of generatorsets from two with-loops in
                with-loop fusion to <n>.
                  (default: 100)

-sigspec <strat>   Specify strategy for specialization of function sigantures:
                      akv: try to infer all values statically,
                      aks: try to infer all shapes statically,
                      akd: try to infer all ranks statically,
```

```
                          aud: do not specialize at all.
                          (default: aks)


MULTI-THREAD OPTIONS:

    -mt             Compile program for multi-threaded execution,
                    e.g. implicitly parallelize the code for non-sequential
                    execution on shared memory multiprocessors.

                    NOTE:
                    The number of threads to be used can either be specified
                    statically using the option "-numthreads" or dynamically
                    upon application startup using the generic command line
                    option "-mt <n>".

    -mtmode <n>     Enable a explicit organization scheme for multi-threaded program
                    execution.
                    Legal values:
                      1: with thread creation/termination
                      2: with start/stop barriers
                      3: with magical new techniques, WARNING: UNDER CONSTRUCTION!!!
                      (default: 2)

    -numthreads <n> Specify at compile time the exact number of threads to be
                    used for parallel execution.

    -maxthreads <n> Specify at compile time only an upper bound on the number
                    of threads to be used  for parallel execution when exact
                    number is determined at runtime.
                      (default: 32)

    -nofoldparallel Disable parallelization of fold with-loops.

    -maxsync <n>    Specify maximum number of fold with-loops to be combined
                    into a single synchronisation block.
                    Legal values:
                      -1: maximum number needed (mechanically infered).
                       0: no fold-with-loops are allowed.
                          (This implies that fold-with-loops are not executed
                           in parallel.)
                      >0: maximum number set to <n>.
                      (default: -1)

    -minmtsize <n>  Specify minimum generator set size for parallel execution
                    of with-loops.
                      (default: 250)

    -maxrepsize <n> Specify maximum size for arrays to be replicated as
                    private data of multiple threads.
                      (default: 250)
                    Option applies to "-mtn" style parallelization only.
```

```
MUTC OPTIONS:

    -mutc_fun_threads Convert all functions to thread functions and use
                      singleton creates

    -mutc_macros      Use mutc macro abstraction interface



BACKEND OPTIONS:

    -minarrayrep <class>
                      Specify the minimum array representation class used:
                        s: use all (SCL, AKS, AKD, AUD) representations,
                        d: use SCL, AKD, AUD representations only,
                        +: use SCL, AUD representations only,
                        *: use AUD representation only.
                      (default: s)


GENERAL DEBUG OPTIONS:

    -d nocleanup      Do not remove temporary files and directories.
    -d syscall        Show all system calls during compilation.
    -d cccall         Generate shell script ".sac2c" that contains C compiler
                      invocation.
                      This implies option "-d nocleanup".


INTERNAL DEBUG OPTIONS:

    -d treecheck      Check syntax tree for consistency with xml specification.
    -d memcheck       Check syntax tree for memory consistency.
    -d sancheck       Check syntax tree for structural consistency.
    -d nolacinline    Do not inline loop and conditional functions.
    -d efence         Link executable with ElectricFence (malloc debugger).


INTERNAL OPTIONS FOR FRED FISH'S DBUG:

    -# t              Display trace information.
                      Each function entry and exit during program execution is
                      printed on the screen.

    -# d              Display debug output information.
                      Each DBUG_PRINT macro in the code will be executed.
                      Each DBUG_EXECUTE macro in the code will be executed.

    -# d,<str>        Restrict "-# d" option to DBUG_PRINT / DBUG_EXECUTE macros
                      which are tagged with the string <str> (no quotes).

    -# <f>/<t>/<o>    Restrict the effect of any Fred Fish DBUG package option <o>
```

```
                    to the range <f> to <t> of sac2c compiler phases.
                      (default: <f> = first compiler phase,
                                <t> = last compiler phase.)
                    All kinds of phases can be specified using a syntax
                    analogous to that of the -b option.


RUNTIME CHECK OPTIONS:

    -ecc            Insert explicit conformity checks at compile time.

    -check [atbmeh]+
                    Incorporate runtime checks into executable program.
                    The following flags are supported:
                      a: Incorporate all available runtime checks.
                      c: Perform conformity checks.
                      t: Check assignments for type violations.
                      b: Check array accesses for boundary violations.
                      m: Check success of memory allocations.
                      e: Check errno variable upon applications of
                         external functions.
                      h: Use diagnostic heap manager.


RUNTIME TRACE OPTIONS:

    -trace [amrfpwstc]+
                    Incorporate trace output generation into executable program.
                    The following flags are supported:
                      a: Trace all (same as mrfpowt).
                      m: Trace memory operations.
                      r: Trace reference counting operations.
                      f: Trace user-defined function calls.
                      p: Trace primitive function calls.
                      w: Trace with-loop execution.
                      s: Trace array accesses.
                      t: Trace multi-threading specific operations.
                      c: Trace runtime enviroment init/exit when
                         using SAC libraries in C programs.

    -utrace
                    Introduce user tracing calls.

RUNTIME PROFILING OPTIONS:

    -profile [afilw]+
                    Incorporate profiling analysis into executable program.
                      a: Analyse all (same as filw).
                      f: Analyse time spent in non-inline functions.
                      i: Analyse time spent in inline functions.
                      l: Analyse time spent in library functions.
                      w: Analyse time spent in with-loops.
```

```
CACHE SIMULATION OPTIONS:

    -cs             Enable runtime cache simulation.

    -csdefaults [sagbifp]+
                    This option sets default parameters for cache simulation.
                    These settings may be overridden when starting the analysis
                    of an application program:
                      s: simple cache simulation,
                      a: advanced cache simulation,
                      g: global cache simulation,
                      b: cache simulation on selected blocks,
                      i: immediate analysis of memory access data,
                      f: storage of memory access data in file,
                      p: piping of memory access data to concurrently running
                         analyser process.
                    The default simulation parameters are "sgp".

    -cshost <name>  This option specifies the host machine to run the additional
                    analyser process on when doing piped cache simulation.
                    This is very useful for single processor machines because
                    the rather limited buffer size of the pipe determines the
                    synchronisation distance of the two processes, i.e. the
                    application process and the analysis process. This results
                    in very frequent context switches when both processes are
                    run on the same processor, and consequently, degrades the
                    performance by orders of magnitude. So, when doing piped
                    cache simulation always be sure to do so either on a
                    multiprocessor or specify a different machine to run the
                    analyser process on. However, this only defines a default
                    which may be overridden by using this option when starting
                    the compiled application program.

    -csfile <name>  This option specifies a default file where to write the
                    memory access trace when performing cache simulation via
                    a file. This default may be overridden by using this option
                    when starting the compiled application program.
                    The general default name is "<executable_name>.cs".

    -csdir <name>   This option specifies a default directory where to write
                    the memory access trace file when performing cache
                    simulation via a file. This default may be overridden by
                    using this option when starting the compiled application
                    program.
                    The general default directory is the tmp directory specified
                    in your sac2crc file.


CACHE SIMULATION FEATURES:

    Simple cache simulation only counts cache hits and cache misses while
    advanced cache simulation additionally classifies cache misses into
```

```
    cold start, cross interference, self interference, and invalidation
    misses.

    Simulation results may be presented for the entire program run or more
    specifically for any code block marked by the following pragma:
        #pragma cachesim [tag]
    The optional tag allows to distinguish between the simulation results
    for various code blocks. The tag must be a string.

    Memory accesses may be evaluated with respect to their cache behaviour
    either immediately within the application process, stored in a file,
    or they may be piped to a concurrently running analyser process.
    Whereas immediate analysis usually is the fastest alternative,
    results, in particular for advanced analysis, are often inaccurate due
    to changes in the memory layout caused by the analyser. If you choose
    to write memory accesses to a file, beware that even for small programs
    to be analysed the amount of data may be quite large. However, once a
    memory trace file exists, it can be used to simulate different cache
    configurations without repeatedly running the application program
    itself. The simulation tool for memory access trace files is called
    'csima' and resides in the bin directory of your SAC installation.

    These default cache simulation parameters may be overridden when
    invoking the application program to be analysed by using the generic
    command line option
        -cs [sagbifp]+
    where the various flags have the same meaning as described for the
    "-csdefaults" compiler option.

    Cache parameters for up to 3 levels of caches may be provided as target
    specification in the sac2crc file. However, these only serve as a
    default cache specification which may well be altered when running the
    compiled SAC program with cache simulation enabled. This can be done
    using the following command line options:
        -cs[123] <size>[/<line size>[/<assoc>[/<write miss policy>]]].
    The cache size must be given in KBytes, the cache line size in
    Bytes. A cache size of 0 KB disables the corresponding cache level
    completely regardless of any other setting.
    Write miss policies are specified by a single letter:
        d: default (fetch on write)
        f: fetch on write
        v: write validate
        a: write around


LIBRARY OPTIONS:

    -linksetsize <n> Specify how many compiled C functions are stored within
                     a single C source file for further compilation and linking.
                     A large number here means that potentially many functions
                     need to be linked to an executable that are actually never
                     called. However, setting the linksetsize to 1 considerably
                     slows down the compilation of large SAC modules/classes
```

```
                     (default: 10)

                     NOTE:
                     A linksetsize of 0 means all functions are stored in a
                     a single file.

    -genlib <lang>   Specify library format when compiling SAC module/class
                     implementations.
                     Supported values for <lang> are:
                       sac: Generate SAC library file (default).
                         c: Generate C object and header files.

                     NOTE:
                     Be careful to use same options for privat heap management
                     (PHM) and profiling for compilation of all modules/classes
                     you are going to link together to a single executable.

                     NOTE:
                     Multithreading is not yet available for C libraries.

    -noprelude       Do not load the standard prelude library 'sacprelude'.


C-COMPILER OPTIONS:

    -g              Include debug information into object code.

    -O <n>          Specify  the C compiler level of optimization.
                      0: no C compiler optimizations.
                      1: minor C compiler optimizations.
                      2: medium C compiler optimizations.
                      3: full C compiler optimizations.
                    (default: 0)

                    NOTE:
                    The actual effects of these options are specific to the
                    C compiler used for code generation. Both the choice of
                    a C compiler as well as the mapping of these generic
                    options to compiler-specific optimization options are
                    are determined via the sac2crc configuration file.
                    For details concerning sac2crc files see below under
                    "customization".


CUSTOMIZATION OPTIONS:

    -target <name>  Specify a particular compilation target.
                    Compilation targets are used to customize sac2c for
                    various target architectures, operating systems, and C
                    compilers.
                    The target description is either read from the
                    installation specific file $SACBASE/runtime/sac2crc or
                    from a file named .sac2crc within the user's home
```

```
                    directory.

    -B <name>       Selects one of the different backends to use. Currently
                    sac2c supports the following backends:

                      c99       default backend that produces c99 code
                      mutc      backend for the mutc extension to C


ENVIRONMENT VARIABLES:

    The following environment variables are used by the SAC compiler suite:

    SACBASE         Base directory of SAC standard lib installation.
    SAC2CBASE       Base directory of SAC installation.



AUTHORS:

    The following people contributed their time and mind to create the
    SAC compiler suite (roughly in order of entering the project):

      Sven-Bodo Scholz
      Henning Wolf
      Arne Sievers
      Clemens Grelck
      Dietmar Kreye
      Soeren Schwartz
      Bjoern Schierau
      Helge Ernst
      Jan-Hendrik Schoeler
      Nico Marcussen-Wulff
      Markus Bradtke
      Borg Enders
      Kai Trojahner
      Michael Werner
      Stephan Herhut
      Karsten Hinckfuss
      Steffen Kuthe
      Jan-Henrik Baumgarten
      Robert Bernecky
      Theo van Klaveren
      Florian Buether
      Torben Gerhards
      Carl A Joslin


CONTACT:

    WWW:    http://www.sac-home.org/
    E-Mail: info@sac-home.org
```

```
BUGS:

    Bugs??  We????

    SAC is a research project!

    SAC tools are platforms for scientific research rather than
    "products" for end users. Although we try to do our very best,
    you may well run into a compiler bug. So, we are happy to receive
    your bug reports (Well, not really "happy", but ...).
```

References

[1] Robert Bernecky, Stephan Herhut, Sven-Bodo Scholz, Kai Trojahner, Clemens Grelck, and Alex Shafarenko. Index vector elimination — making index vectors affordable. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages*, volume 4449/2007 of *Lecture Notes in Computer Science*, pages 19–36, Berlin / Heidelberg, 2007. Springer.

[2] C. Grelck, K. Hinkfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In Frank Huch A. Butterfield, Clemens Grelck, editor, *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05, Selected Papers*, volume 4015 of *LNCS*, pages 178–195. Springer, 2006. to appear.

[3] C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proc. of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, Selected Papers*, volume 1868 of *LNCS*, pages 77–95. Springer, 2000.

[4] Clemens Grelck, Stephan Herhut, Chris Jesshope, Carl Joslin, Mike Lankamp, Sven-Bodo Scholz, and Alex Shafarenko. Compiling the Functional Data-Parallel Language sac for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zurich, Switzerland*, 2009.

[5] Stephan Herhut, Carl Joslin, and Sven-Bodo Scholz. Compiling the functional data-parallel language sac for the self-adaptive virtual processor architecture. Technical Report 482, School of Computer Science, University of Hertfordshire, 2008.

[6] Chris R. Jesshope. mutc - an intermediate language for programming chip multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, pages 147–160, 2006.

[7] Alexei Kudryavtsev, Daniel Rolls, Sven-Bodo Scholz, and Alex Shafarenko. Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In *10th International Conference on Parallel Computing Technologies*. accepted, 2009.

[8] S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

[9] S.-B. Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[10] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra. An implementation of the sane virtual processor using posix threads. *J. Syst. Archit.*, 55(3):162–169, 2009.