Steffen Kuthe

A Hybrid Shared Memory Execution Model for SAC

Diplomarbeit

am Institut für Softwaretechnik und Programmiersprachen
Direktor: Prof. Dr. Walter Dosch
Technisch-Naturwissenschaftliche Fakultät
Universität zu Lübeck

Betreuer:
Dr. Clemens Grelck

Januar 2005
Ich versichere, dass ich diese Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Lübeck, 31. Januar 2005

(Unterschrift)
Contents

1 Introduction ............................................. 1

2 SAC - Single Assignment C .............................. 3
   2.1 A short survey ........................................ 3
   2.2 Arrays .................................................. 4
   2.3 The with-loop construct .............................. 4
   2.4 Memory management in SAC ......................... 7

3 Models of parallel execution ............................ 9
   3.1 The fork/join model .................................. 9
   3.2 The SPMD model ....................................... 11
   3.3 The BSP model ......................................... 13

4 Multithread execution model of SAC ..................... 17
   4.1 Basic ideas ............................................. 17
   4.2 Improved fork/join model ............................ 19
   4.3 Reducing synchronisations ............................ 19

5 Ideas of a new execution model ......................... 25
   5.1 Code replication ....................................... 25
   5.2 Execution modes ....................................... 26
   5.3 Rearranging code ..................................... 29
   5.4 Communication structure ............................. 33
6 Realisation of the new execution model 41
   6.1 Execution modes ........................................... 41
       6.1.1 Any threaded execution mode .......................... 41
       6.1.2 Handling functions .................................... 42
   6.2 Rearranging assignments .................................. 43
       6.2.1 Basics of graph theory ................................. 43
       6.2.2 The rearranging algorithm ............................. 44
       6.2.3 Step 1 - Reducing execution modes changes ........ 46
       6.2.4 Step 2 - Preorder nodes of the same set ............ 48
       6.2.5 Step 3 - Fixing the order ............................. 50
       6.2.6 Building execution cells ............................... 51
   6.3 Split-Phase Communication ................................ 53
       6.3.1 Synchronisation of data dependencies ................. 54
       6.3.2 Waiting for the allocation of shared memory ........ 57
   6.4 Consequences for the memory management ............... 58

7 Optimisation opportunities 63
   7.1 Topological synchronisations .............................. 63
   7.2 Balanced execution of st-cells ............................ 65
   7.3 Distributed memory allocation ............................. 66
   7.4 Parallelisation beyond function borders ................ 66

8 Conclusion 69

Bibliography 71
List of Figures

2.1 Examples of SAC’s representation of arrays. .................. 4
2.2 The simplified syntax of with-loops. ............................. 5
3.1 The pattern of fork-join. ........................................... 10
3.2 The pattern of SPMD. .............................................. 12
3.3 The pattern of a BSP superstep. .................................. 13
4.1 Code generation scheme for a genarray with-loop. ............. 18
4.2 Comparison between pure and improved fork/join model. ....... 20
4.3 Scheme of an enhanced stop barrier to reduce overhead. ....... 21
4.4 Execution with and without superfluous synchronisation. ...... 22
5.1 Schematical execution of code without and with replication re-
spectively. ............................................................ 27
5.2 Forking additional threads in multi-threaded execution leads to
a strong increase of threads. ....................................... 28
5.3 Schematical code execution of the new execution model. ..... 30
5.4 Schematical code execution with a superfluous synchronisation
because of an et-assign. ............................................ 32
5.5 The schematical code execution of the rearranged code shows
no more synchronisation. .......................................... 32
5.6 Schematical code execution with short distances between mem-
ory allocation and usage. ......................................... 35
5.7 Schematical code execution of rearranged code shows less time
to wait before using memory. ..................................... 35
5.8 SAC code with marked execution cells. .......................... 36
5.9 Comparison between strict barrier synchronisation and split-phase synchronisation. ........................................ 37
5.10 Example of the usage of split-phase communication between mt- and st-cells. .................................................. 39

6.1 Function and its data flow graph representation. ................. 44
6.2 Example data flow graph to illustrate the rearranging algorithm. 45
6.3 The first step of the rearranging algorithm. ....................... 47
6.4 The data flow graph after the first algorithm step. ............... 48
6.5 The second step of the rearranging algorithm. .................... 49
6.6 The result of the second algorithm step. ........................ 50
6.7 The third step of the rearranging algorithm. ...................... 52
6.8 The result of the third algorithm step. .......................... 52
6.9 The resulting data flow graph with marked execution cells. ... 53
6.10 Example of the wait barrier’s functionality. ..................... 56
6.11 Extended array of flags to be used with fold-with-loops. ....... 56
6.12 Global accessible buffer to broadcast memory addresses. ...... 57
6.13 Mt-descriptor to support two layer reference counting. ....... 59
6.14 Improved mt-descriptor to support two layer reference counting, including padded spaces to avoid the effects of false sharing. ... 60

7.1 Comparison of split-phase and topological synchronisation. ... 64
7.2 Potential of parallelisation beyond function borders. .......... 67
Listings

2.1 Example of a with-loop with genarray operation. 6
2.2 Example of a with-loop with modarray operation. 6
2.3 Example of a with-loop with fold operation. 7
4.1 Code which leads to a superfluous synchronisation during multithread execution. 21
4.2 Generated SPMD sections. 23
4.3 Merged SPMD sections to reduce synchronisations. 24
5.1 Simple scalar operation prohibits the computation of two with-loops without synchronisation. 26
5.2 Code example with annotated execution modes. 30
5.3 Code arrangement that causes several synchronisations. 31
5.4 Rearranged code of Listing 5.3 that causes less synchronisations than before. 31
5.5 Code arrangement which is impractical for multi-threaded execution due to the adjacencies of memory allocation and usage. 34
5.6 Rearranged code of Listing 5.5 that is more practical for multi-threaded execution. 34
5.7 Three with-loops that can not be executed without a synchronisation in between within the existing execution model. 36
5.8 Listing 5.7 with added instructions to perform split-phase synchronisation. 38
7.1 Example to show the potential of partial synchronisation. 63
Chapter 1

Introduction

The functional programming language Single Assignment C or SAC, is developed in particular for high-performance computation of arrays in a functional environment. The language is free of side-effects and includes implicit memory management. SAC's design features potentially complex map- and fold-like operations on arrays, the with-loops. These constructs are well-suited for concurrent execution. Previous works [Sch00, Gre01, Gre03] have shown this potential of with-loops on shared memory multiprocessor systems.

The idea of the existing multi-threaded execution model is to execute each with-loop in parallel separately using a fork/join pattern. The memory to store the result of such a with-loop is allocated before forking and after joining by a special thread. Adjacent with-loops can be combined and executed within only one parallel execution to save superfluous synchronisations between them. These possibilities of parallelisation exploited so far achieve good results [Gre02a, GS03], but they are limited due to their concept of execution. Data dependencies between with-loops prohibit their combined execution and parallel execution is restrain only to with-loops.

This thesis is about designing a novel execution model to execute SAC programs in parallel on shared memory multiprocessor systems. The idea is to enlarge the regions of parallel execution. Therefore, each assignment is classified with respect to its execution possibility as exclusive-, single- or multi-threaded. Multi-threaded assignments are subdivided into parts and executed by multiple threads, whereas single-threaded and exclusive-threaded assignments are executed by one thread. In contrast to single-threaded assignments, exclusive-threaded assignments must not be executed in parallel to multi-threaded assignments. Several assignments have no determined execution mode. They can
be executed either sequentially or replicated in parallel. This is determined corresponding to the environmental assignments.

Starting from these classified assignments, so-called execution cells are built by collecting consecutive assignments of identical execution modes. Hence, the result is the partition of the SAC program into several areas of identical execution modes. This leads into a hybrid mixture between the SPMD and the fork/join model. Like in [CLS90], it is decided for each assignment, which way of execution is the best.

The new execution model enables even the occurrence of data dependent assignments within the same execution cell. This is realised via the use of split-phase synchronisation to guarantee correct data handling. The fact of larger areas in multi-threaded execution prohibits the allocation of all memory for such an area beforehand. Hence, the dynamic memory management is adapted to support memory management operations during execution in parallel.

Each changing into an exclusive-threaded execution cell requires a time-consuming synchronisation. Via rearranging the program code, the number of these changes into exclusive-threaded execution cells is reduced. This rearranging algorithm, based on graph theory, maximises the distance in execution of data dependent assignments, as well. This increases the probability of a computation being finished by the slowest thread when the fastest thread needs its result.

Further on, some optimisation opportunities of the new model are discussed. Topological synchronisations [SM96] to decrease thread idle time and parallelisation beyond function borders to eliminate synchronisations are shown.

The remainder of the thesis is organised as follows. Chapter 2 briefly introduces the functional programming language SAC. Current established techniques of execution models, like the SPMD or the fork/join model, are presented in Chapter 3. Chapter 4 describes the existing parallelisation scheme of SAC. The new hybrid shared memory execution model is presented in detail in Chapter 5. Afterwards, in Chapter 6, techniques to realise this innovative model are shown. After some more notes in Chapter 7, that show additional potential to optimise the new parallel execution model, Chapter 8 concludes the work.
Chapter 2

SAC - Single Assignment C

2.1 A short survey

SAC [Sch96, Sch97, Sch98] is a programming language whose design is focused on the needs of numerical applications. It is purely functional with a syntax similar to that of ANSI-C [KR88]. Compared to the imperative control flow in C, all language constructs, like assignments, loops and conditionals, are interpreted in a functional way. Loop constructs are considered as tail-recursive functions and, in their syntactical positions, replaced by the respective function calls [Sch03]. SAC does not support pointers or global variables. It also uses call-by-value parameter passing and avoids the call-by-reference way. These restraints guarantee the absence of side-effects.

This language design supports a large amount of different possibilities to optimise the code, outperforming the possibilities of strict C-code, like the common subexpression elimination, as described in [MW01]. Thus, each call of a function with identical parameters produces identical results, independent of the current part of the program. This allows to compute a function call just once and replace all identical occurrences with the computed function result.

According to the similar syntax of C and SAC, the change from the imperative to the functional way of programming is easy for an imperative programmer. Additionally to the well-known structures of C, SAC brings some innovations (relating to C), like functions with any count of return parameters. SAC supports arrays as first class objects and provides high-level, shape- and dimension-invariant operations on them. Furthermore, SAC features a very flexible module- and I/O-system [Gre96].
Array | Shape vector | Data vector
--- | --- | ---
$A_1 = (6, 28, 42)$ | [3] | [6, 28, 42]
$A_2 = \begin{pmatrix} 
6 & 28 & 42 \\
42 & 28 & 6
\end{pmatrix}$ | [2, 3] | [6, 28, 42, 42, 28, 6]

*Figure 2.1:* Examples of SAC’s representation of arrays via data and shape vectors.

### 2.2 Arrays

On the conceptual view all values in SAC are arrays. Arrays are represented by two vectors, a *shape vector* which specifies the number of elements per axis, and a *data vector* containing the array’s elements in a row-major unrolling. Figure 2.1 shows some examples for the representation of arrays in SAC.

For easy handling of arrays, SAC brings several primitive functions to get and set the shape, the data or manipulate even single elements of arrays. Additionally SAC features a powerful language construct for array-manipulation - the *with-loop* construct, which is the topic of the next section.

### 2.3 The with-loop construct

With-loops represent a SAC-specific language construct for the definition of aggregate array operations. These map- and fold-like operations are free of side-effects and are well-suited for parallel execution (see Section 4.1). The declaration of a with-loop is independent of an array’s shape. This independence enables universal dimension-invariant programming, i.e. not only extents of argument or result arrays can vary, even their dimensionality itself.

The with-loops offer the possibility to provide the same functionality of array operations as provided by other languages like APL [Ive62], implemented in SAC itself, with almost no loss of generality [GS99]. In addition, the with-loop makes it easy to adapt code of other array languages to SAC. Figure 2.2 outlines the simplified syntax of with-loops. See [Sch98] or [Kre98] for more details.

In general a with-loop consists of two parts: a *generator* and an *operation.*
2.3. THE WITH-LOOP CONSTRUCT

\[
\begin{align*}
Expr & \Rightarrow \ldots \\
& \mid \text{WithExpr}
\end{align*}
\]

\[
\begin{align*}
\text{WithExpr} & \Rightarrow \text{with} \ ( \text{Generator} \ ) \ [ \ \{ \ \text{Assign} \}^* \ ] \ \text{Operation} \\
\text{Generator} & \Rightarrow \text{Expr} \ \text{Relop} \ \text{Id} \ \text{Relop} \ \text{Expr} \\
\text{Relop} & \Rightarrow < \mid \leq \\
\text{Operation} & \Rightarrow \text{genarray} \ ( \ \text{Expr} , \ \text{Expr} \ ) \\
& \mid \text{modarray} \ ( \ \text{Expr} , \ \text{Expr} , \ \text{Expr} \ ) \\
& \mid \text{fold} \ ( \ \text{FoldFun} , \ \text{Expr} \ [ , \ \text{Expr} ] \ )
\end{align*}
\]

\[
\begin{align*}
\text{FoldFun} & \Rightarrow \text{Id} \mid \text{FoldPrf} \\
\text{FoldPrf} & \Rightarrow + \mid * \mid \&\& \mid \mid \mid \text{min} \mid \text{max}
\end{align*}
\]

**Figure 2.2:** The simplified syntax of with-loops.

**The with-loop generator** The generator defines lower and upper bounds for a set of index vectors and an index variable, which represents an element of this set. This index variable makes it possible to reference single elements, e. g. in the with-loop operation. The optional filter is for restricting the set of index vectors. For example, with \( lo, up, st \) and \( wi \) denoting expressions that evaluate to vectors of length \( n \),

\[
(lo \leq iv < up \ \text{step} \ st \ \text{width} \ wi)
\]

specifies the set of index vectors

\[
\{ \{ iv | i \in [0, \ldots, n-1] : lo_i \leq iv_i < up_i \land (iv_i - lo_i) \mod st_i < wi_i \} \}
\]

**The with-loop operation** There are three different operations, which give the meaning of a with-loop. They are defined as follows, where \( expr \) denotes some SAC expression, \( shp \) and \( idx \) denote expressions which evaluate to vectors, \( array \) denotes an expression which evaluates to any kind of array and \( fold_op \) is the name of a binary associative and commutative function with its neutral element \( neutral \):

- \( \text{genarray}(shp, \ expr) \)
  It creates an array with the shape \( shp \). All its elements, that are in the set of the generator, get the value of \( expr \). All other elements get the value 0.
$A = \textbf{with}\ ([1,0] <= \text{iv} < [4,7])$
$\text{genarray}([4,7], \text{iv}[0] + \text{iv}[1]);$

\textbf{Listing 2.1:} Example of a with-loop with genarray operation.

For instance, $A$ in Listing 2.1 is a $4 \times 7$ array. All its rows, skipping the top one, have the sum of the index vector entries in their elements.

$$A = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 3 & 4 & 5 & 6 & 7 & 8 \\
3 & 4 & 5 & 6 & 7 & 8 & 9
\end{pmatrix}$$

- $\text{modarray}(\text{array, idx, expr})$
  It creates an array with the same shape as $\text{array}$. All its elements that are in the set of the generator get the values of $\text{expr}$. Each other element get the value of the given array $\text{array}$ at the index vector position $\text{idx}$.

$B = \textbf{with}\ ([0,2] <= \text{iv} < [3,5])$
$\text{modarray}(A, \text{iv}, 42);$

\textbf{Listing 2.2:} Example of a with-loop with modarray operation.

$B$ in Listing 2.2 is an array with the identical shape of $A$. It has the same elements as $A$, too, except for a $3 \times 3$ area, where all elements have 42 as value.

$$B = \begin{pmatrix}
0 & 42 & 42 & 42 & 0 & 0 \\
1 & 2 & 42 & 42 & 42 & 6 & 7 \\
2 & 3 & 42 & 42 & 42 & 7 & 8 \\
3 & 4 & 5 & 6 & 7 & 8 & 9
\end{pmatrix}$$

- $\text{fold}(\text{fold\_op, neutral, expr})$
  It computes the expression $\text{expr}$ for those elements, that are in the set defined by the generator. Afterwards, it folds these expressions by using the fold function $\text{fold\_op}$ and $\text{neutral}$ as its neutral element. The function $\text{fold\_op}$ must be commutative and associative. When using a primitive function, the declaration of $\text{neutral}$ is obsolete.
2.4  MEMORY MANAGEMENT IN SAC

\begin{verbatim}
prod = with ([0, 0] <= iv < [4, 7])
    fold (*, A[iv]);
\end{verbatim}

Listing 2.3: Example of a with-loop with fold operation.

For example, in Listing 2.3, the with-loop computes the product of all elements of the given array $A$.

\[ prod = 36870930432000 \]

2.4 Memory management in SAC

One feature of SAC is its dynamic memory management in form of reference counting [Wol95, GT04]. Thus, the programmer need not to take care about memory issues and can concentrate on writing correct programs.

The implementation of the memory management is based on reference counting. Each array has an attributed reference counter. If the reference counter decreased to zero, the array is not needed anymore. Hence, its memory can be deallocated.

As an improvement, the memory management supports memory reuse. Thus, allocated memory which is not needed anymore is reused before new memory is allocated.

All attributes of an array, the reference counter and the shape vector, are encapsulated in a descriptor. Each array has its own attributed descriptor.
CHAPTER 2. SAC - SINGLE ASSIGNMENT C
Chapter 3

Models of parallel execution

This chapter describes three different execution models for parallel execution. They stand representatively for a multiplicity of existing models. Starting with the currently used fork/join model (Section 3.1), the universal SPMD model is described in Section 3.2. The last section shows an interesting mixture of both models, the BSP model (Section 3.3). A more detailed overview over several execution models for parallel execution can be seen in [Keß00].

3.1 The fork/join model

In the fork/join model [Con63, DvH66], one initial thread executes instructions sequentially, until a parallel construct is encountered. To compute this construct, the initial (master) thread forks several (worker) threads. They execute the body of this construct together with the initial thread and join after finishing its computation. The execution proceeds sequentially, until reaching the next parallel construct. This leads to a permant changing between sequential and parallel execution, at which the number of parallel working threads can potentially differ from time to time. Hence, during a fork/join execution one thread works all time, all the other threads work part-time. For an abstract overview, Figure 3.1 shows the schedule of a typical fork/join execution of a program.

There is no kind of communication between the single threads within the parallel execution. They get their work at the beginning via the fork-mechanism and return their result through the joining. All threads have to return a result, at least to signal the finish of their work. This is often used, whenever each thread just gets a part of a huge global data-object and it has to modify its
part, at which it would not be advisable to copy all data for returning it.

Besides the usage of this model in SAC for providing implicit shared memory multiprocessor support [Gre01], it is widely used for shared memory architectures, e.g. in OpenMP [DM98], where the programmer can annotate parallelisation explicitly via pragma statements.

The fork/join execution model has many appealing features, compared to the SPMD model (Section 3.2). It avoids the danger of deadlocks by restricting the communication in time (only during fork and join) and direction (one-to-many during fork, many-to-one during join). Due to the dynamic change of the number of threads, the model benefits sharing the machine with other programs in execution and makes it very suitable for all kind of unfold/fold algorithms.
Since only small parts differ from a sequential version, fork/join as compiling scheme is relatively easy to implement, especially if the concerning programming language supports fitting constructs, like with-loops in SAC. Each of the computations in a with-loop is independent of each other, which makes with-loops very suitable for this execution model. Hence, a with-loop can be used as a starting point for concurrent execution. Using one fork/join pattern for more than one with-loop leads to several problems, e.g. because of data dependencies, which is discussed in Chapter 4.

Although threads as “light-weight” processes are faster to fork and to join than operating system processes, the permanent switch between single and concurrent execution has an overhead, e.g. by allocating memory, copying control data and swapping registers. Long latencies of an interconnection network restrain the usage of fork/join to shared memory multiprocessor machines. Distributing and collecting huge data constructs would sweep off all benefit of parallel execution.

A problem to implement performant compiling schemes is the restricted communication scheme of fork/join. Many numerical applications, like the approximation of the temporal temperature distribution in solids, are well suited for parallelisation. Partitioning these problems leads into several blocks, whose computation needs data from surrounding blocks. This data must be read in a well defined state to avoid errors in computation. Adapting these algorithms produces lots of fork and join actions (or much more space in memory alternatively) and thus much overhead. This counts for every algorithm that needs much communication.

### 3.2 The SPMD model

SPMD stands for Single Program / Multiple Data stream, which is a popular addition to the MIMD (multiple instruction multiple data) system of Flynn’s taxonomy [Fly66]. In the SPMD model [Jor86, DGNP88] each thread executes the same instruction stream. In spite of this constraint it is possible to reach different behaviour of individual threads by using a unique thread ID for each thread. Figure 3.2 shows an abstract overview of the pattern of the SPMD execution model.

Characteristic for this execution model is the alternate appearance of replicated and parallel parts of the code. Replicated means each thread computes exactly the same instructions redundantly. Parallel stands for the execution of different instructions and identical instructions with different parameters, respectively.
Hence, all threads work all time during an SPMD execution, as long as they
do not wait for data of other threads.

Another characteristic of SPMD is its complete freedom of communication
possibilities. Beginning with simple point-to-point communication, any kind
of communication between any selection of threads is possible (e.g. broadcast,
reduce). There are different libraries like the message passing interface library
MPI [GLS94] or the parallel virtual machine PVM [Sun90], that provide many
of these communication schemes.

Using the SPMD model has some interesting advantages. Due to its complete
freedom of communication possibilities, it is very flexible to be adapted to
many kinds of algorithms. Only algorithms with an increasing number of
threads can not be mapped on this execution model easily. Another advantage
of SPMD is its suitability for distributed memory. Due to the absence of any
kind of global memory, SPMD programs can be executed on shared memory
machines as well.

Due to the flexibility of the SPMD model in communication and execution
a compilation scheme using SPMD can act well adapted to the depending
language. SPMD allows many highly specialised compiling schemes for several
possible code arrangements, e.g. in SAC the sequence of two with-loops may
have a specialised compiling scheme.
3.3. THE BSP MODEL

![Diagram of BSP superstep]

Figure 3.3: The pattern of a BSP superstep.

One problem using SPMD as compiling scheme for SAC is the absence of global memory. To compute a `genarray-with-loop` or a `modarray-with-loop` in parallel, the memory for the result has to be allocated. In SPMD a thread has to allocate the memory of the `with-loop`'s part it processes by itself. If another thread needs some values of this part, it has to request for it. This procedure produces much overhead on a shared memory machine.

The possibility of different communication schemes is another pitfall of SPMD. All compiling schemes using the SPMD model must guarantee their freedom of deadlocks, especially in nested occurrences.

It would be a big challenge to use the SPMD model as a new execution model for SAC, using all its communication potential to have a execution model, suitable for many architectures. Also its suitability for distributed memory would be very interesting for future work and focus of SAC, but the development of a corresponding memory management system to support this model is far beyond this thesis.

### 3.3 The BSP model

The BSP (bulk synchronous parallel) model, as introduced by Valiant [Val90, SHM97], structures a parallel computation of threads into a sequence of so-called supersteps. Each superstep (see Figure 3.3) consists of three phases:
1. Local computation, which uses only local stored variables (i.e. located in the memory of the respective processor),

2. Global communication, which moves data as messages between the processors,

3. Global barrier synchronisation, which waits for all processors to complete their computation and communication actions. Afterwards, it makes the transmissions of the global communication available in the memory of the destination processors.

This structure results in a mixture between the fork/join (restricted communication pattern) and the SPMD model (fixed number of always online threads).

Like SPMD, BSP is independent of its target architectures. Due to the existence of a BSP library for many programming languages, it is independent of a special language as well. Due to BSP's definite structured programming and workflow scheme, a compiling scheme would be easy to write.

As fork/join programs, execution via BSP looks much the same as sequential execution. This makes it easy to adapt a sequential program to this model. Its structure leads to a conceptual prevention of deadlocks and also to a secure behaviour of the program.

Another feature of BSP is the possibility of runtime estimation. The performance of a program on a given architecture is predictable because of an existing cost model, at which among other things the restricted communication scheme is included.

This restricted communication leads into the same problems like in the fork/join model. Due to the fact of no communication during the computing phase of BSP, a superstep could only be as long as the shortest distance between two communications in the SPMD model. To enable larger steps, one must adapt the algorithm by reordering or dividing parts. Thus, the BSP model is not easy to adapt on many algorithms. As the SPMD model, BSP is very inflexible concerning all algorithms, that needs a dynamic changing number of threads, too.

The global barrier synchronisation results in another problem. As mentioned in [McC96], it is very inflexible for structuring parallel programs. It also leads to a quadratic increase of time to synchronise by linear rise of the number of threads. A solution for this problem are nested sub-supersteps, as in NestStep [Ke600], but this leads to an even more complex model.

BSP is well suited for processing with-loops, but as the fork/join approach it has its limits to process more than one with-loop without any synchronisation
in-between because of frequently appearing data dependencies. Hence, a new multithread execution model for SAC can take some design elements of the BSP-superstep model as a starting point, but it must overcome the mentioned limits to achieve good results.
Chapter 4

Multithread execution model of SAC

4.1 Basic ideas

The old multithread execution model of SAC [Sch00, Gre01, Gre03], based on the fork/join approach, aims on a shared memory machine as basic architecture. The implementation is realised via POSIX-Threads [Ins95] to keep it independent of a particular operating system.

As mentioned in Section 2.3, SAC supports with-loops as powerful constructs to generate and modify large arrays. All variants of the with-loop are well suited for non-sequential execution. Each with-loop represents a large number of (probably complex) element-wise computations, that are independent of each other.

As mentioned in Section 2.3, all functions operating on arrays are implemented in SAC itself. Hence, the execution of a typical SAC-program takes place mostly in with-loops. Thus, improving the computation of with-loops will improve nearly the whole program.

Figure 4.1 shows the compilation scheme of a genarray-with-loop. Whenever the initial thread reaches a genarray-with-loop, it allocates the memory for the result array first, which is assigned to the previously not used variable tmp. As the next step the base address of the result array tmp, its shape and the numerical arguments of the operation are copied into a global container cont. Afterwards, a group of worker threads is created.

Each worker thread uniformly executes the same code, only parameterised by a unique ID to identify itself. First of all, it has to build a local copy of the
Figure 4.1: Code generation scheme for a genarray with-loop.

content of cont to set up an appropriate execution environment. This way of copy operations to and from a specific part of the memory corresponds to send and receive operations on other platforms. Afterwards, each worker thread can determine its subspace of the total index space by using its ID as a parameter of the scheduling function Subset. Subset returns a set of individual index vectors, depending on a scheduling scheme, that is specified at compile time [Gre01, End02]. For each element of its index space idxsetᵢ the worker thread i computes the corresponding operation and writes it to the result array. After its computation, each worker thread terminates to signal its completion to the master thread. As soon as all worker threads have terminated, the master thread continues with sequential program execution.

The code generation of a modarray-with-loop is quite similar. It can be mapped on the code generation of a genarray-with-loop. The value of the initialising array is used as default value compared to 0 at the genarray-with-loop.

For fold-with-loops, each thread owns a local accumulation variable accu, initialised by the neutral element of the fold function. After finishing its computation, the thread returns accu's value to the master thread and terminates. The master thread receives all those partial results. Afterwards, it computes the global fold result.
4.2 Improved fork/join model

The existing implemented multithread execution model of SAC (as described in Section 4.1) gets two essential enhancements. Firstly, it gets an advanced fork/join model to reduce the startup time of multithread execution described in this section. Secondly, the reduction of synchronisation operations is described in Section 4.3.

The pure fork/join model as described in Section 3.1 is based upon recurrent creation and termination of threads, as shown on the left hand side of Figure 4.2. Although thread creation is not as time-consuming as process creation, it still consumes time and resources.

To avoid this overhead, all worker threads are created once at program startup. They stay alive through the whole program execution and terminate at the end of the program (as illustrated on the right hand side of Figure 4.2). Directly following its creation, each worker thread reaches a start barrier. This lets the thread wait until the master thread encounters the first with-loop that is executed in parallel.

After preparing parallel with-loop-execution as shown in the section before, the master thread lifts the barrier of the worker threads, instead of creating them from the scratch. Now, each worker thread performs its concurrent work. Afterwards, it reaches a stop barrier and, with nothing else to do, it can immediately proceed to the next start barrier (or, in case of program termination, terminate).

This combination of a stop barrier and a corresponding start barrier represents a full barrier synchronisation. This causes well known overhead, which leads to bad scalable code at growing number of threads [HS98]. To minimise this problem, the stop barrier is organised as a tree-like structure with synchronisations in pairs, as shown in Figure 4.3. Starting synchronisation with its neighbour, one of this two threads synchronises with their neighbour-pair and so on, until finally the last active thread of one half of the threads synchronises with the last active thread of the other half [Gre03].

4.3 Reducing synchronisations

A second approach to improve the performance of the old multithread execution model is the expansion of regions executed in parallel. Up to now, each region executed in parallel is bound to exactly one with-loop, even in situations, where this is obviously unnecessary. Listing 4.1 shows an example of
Figure 4.2: Comparison between pure and improved fork/join model.
4.3. REDUCING SYNCHRONISATIONS

![Diagram of an enhanced stop barrier to reduce overhead.]

**Figure 4.3:** Scheme of an enhanced stop barrier to reduce overhead.

\[ A = \textbf{with} \ [0, 0] \leq [i, j] \leq [1000, 1000]) \]
\[
\text{genarray}( [1000,1000], i \times j); \]

\[ B = \textbf{with} \ [0, 0] \leq [i, j] \leq [1000, 1000]) \]
\[
\text{genarray}( [1000,1000], i + j); \]

**Listing 4.1:** Code which leads to a superfluous synchronisation during multi-thread execution.

two \textbf{with}-loops, whose computation is totally independent of each other. The compilation scheme would produce a start and stop barrier for each of the two \textbf{with}-loops, respectively (Figure 4.4, left hand side). The synchronisation in-between these \textbf{with}-loops is superfluous for a correct program execution.

To realise this elimination of needless synchronisations, each \textbf{with}-loop is embedded during the compilation into a meta construct, called \textit{SPMD section}. Each SPMD section includes three sets of variables to keep data dependencies in mind. These sets are

- \textit{IN}, for all variables declared somewhere before the section and used within,
- \textit{OUT}, for all result variables of \textbf{fold-with}-loops of the section and
- \textit{INOUT}, for all result variables of the section’s \textbf{genarray-with}-loops and \textbf{modarray-with}-loops. The memory of this \textit{INOUT} variables must be allocated beforehand.
Figure 4.4: The generated execution scheme for the example of Listing 4.1 includes superfluous synchronisation operations (left hand side), that can be reduced via the SPMD sections (right hand side).
Listing 4.2 shows the SPMD sections, that were created from the with-loops of the example of Listing 4.1. These initial SPMD sections are the starting point to merge neighbouring ones, if there exist neither data dependencies nor anti-dependencies between them (see [Gre01, Gre02b] for a more detailed overview over the merging algorithm). This results into bigger SPMD sections, which reduces the number of start/stop barrier pairs by one per merging operation (Listing 4.3).

The arising execution scheme shows less overhead than before (Figure 4.4, right hand side). In addition to the eliminated synchronisation operation another positive effect occurs. The combined runtime of the two with-loops is not the sum of the slowest thread runtime of the first with-loop with the slowest thread runtime of the second one anymore. Now, the combined runtime is the runtime of the thread, which needs the most time to compute both with-loops. This leads never to a longer runtime, but often to a shorter one.
\begin{verbatim}
SPMD (IN    = { },
      INOUT = {A, B},
      OUT   = { })

  { 
    A = with ([0,0] <= [i,j] < [1000,1000])
         genarray ([1000,1000], i * j);

    B = with ([0,0] <= [i,j] < [1000,1000])
         genarray ([1000,1000], i + j);
  }
\end{verbatim}

\textbf{Listing 4.3:} The merged SPMD sections for the example of Listing 4.2.
Chapter 5

Ideas of a new execution model

The design of a new execution model must consider four different but connected issues. First of all, code replication as the basic idea of the new execution model is described in Section 5.1. Afterwards, different execution modes in consequence of the code replication are introduced in Section 5.2. The chapter continues with ideas about rearranging code (Section 5.3) before Section 5.4 presents a new communication structure.

5.1 Code replication

The existing model to execute SAC in parallel has reached a stable status and is serviceable. By using the fork/join approach it was possible to develop understandable schemes for compilation. These schemes yield good results in speedup despite their simplicity [Gre01, Gre02a, GS03]. Nevertheless, the execution model is subject to restrictions.

One limitation of the existing multi-threaded execution model is the restricted possibility of merging SPMD sections. Simple scalar operations in between consecutive with-loops executed in parallel, prohibit the continuous computation of the with-loops in a multi-threaded environment. An example for such a situation is shown in Listing 5.1. This leads to superfluous, time-consuming synchronisations.

Even if two consecutive with-loops existed in the user’s SAC code, the consecutiveness would not be guaranteed to survive during compilation until the SPMD sections appear. Several phases of the compiler, especially the optimisations [Sie95, MW01], add, manipulate or delete code. Thus, a solution


```
A = with([0, 0] <= [i, j] < [1000, 1000])
genarray([1000, 1000], i * j + k);
k = 2 * k;
B = with([0, 0] <= [i, j] < [1000, 1000])
genarray([1000, 1000], i * j + k);
```

**Listing 5.1:** Simple scalar operation prohibits the computation of two with-loops without synchronisation.

must be found to enable execution of those operations without any superfluous synchronisation.

The solution to this problem must enable an overall parallel execution for any possible arrangement of code. It must consider all kinds of data dependencies while avoiding any additional insertion of synchronisations and restarts of the parallel environment. This solution is the replication of code.

The functionality of replication is as follows. Imagine each thread uses identical resources, usually a processor of the same type. Assuming a piece of code cannot be scheduled in parallel. Therefore, it does not matter for the runtime of the program, if this piece of code is executed by one thread (the master thread) or by all threads (master thread and worker threads) at the same time. Thus, computing this code in a replicated manner enables the worker threads to continue their execution going on to the next multi-threaded with-loop without a synchronisation barrier in between (as shown in Figure 5.1).

Furthermore, not only synchronisation overhead is reduced. The replication of code can gain even an advantage under certain circumstances due to the reduction of communication. If the worker threads require the results of this code (as in Figure 5.1), the master thread does not need to broadcast them because the worker threads have already computed the results by themselves.

### 5.2 Execution modes

It is not always possible to replicate code because of the appearance of several problems. These problems will be addressed shortly in the following. Afterwards, it will be shown how the new execution model addresses them.
5.2. EXECUTION MODES

Figure 5.1: Schematic execution of the code in Listing 5.1 without (left hand side) and with replication (right hand side), respectively.

One problem are I/O operations [GS95]. It must be guaranteed that neither outputs, which shall be conducted once, nor inputs, which shall receive something once, must not be replicated.

Other candidates to be computed by just one thread are unique variables in SAC. Their uniqueness property [SBvE+93], the quality of being referenced exactly once, prohibits their replication. Additionally, SAC’s memory management includes instructions that can only be executed by one thread alone as well, like the memory allocation of shared arrays. This allocation is done by the master thread and must not be replicated to avoid memory inconsistencies.

Another difficulty is the handling of library functions. Due to the fact that this external functions can include any kind of compiled SAC-code (or even C-code using the C-interface of SAC), in particular, it may contain some parallel expansion. This connotes that one thread can call an internally parallelised function. The execution of this function in replication produces a strong increase of the number of threads (as shown in Figure 5.2). The number of threads is usually optimally mapped to the machine (n threads for n processors and processor cores at multi core CPUs, respectively). The entire number of threads as a result of such external functions would be much higher than desired. This leads to an ineffective time-consuming execution of several threads per processor using time-slicing. Thus, it must be possible for some assignments, not only being executed by exactly one thread, but also being executed as the only active thread in the system.
Figure 5.2: Forking additional threads in multi-threaded execution leads to a strong increase of threads.

Putting these things together, we can distinguish between three kinds of different execution environments. Each assignment, including loops and conditionals, gets a tag during compilation to identify its execution mode.

1. An assignment that will be executed by exactly one thread and all other threads are deactivated during its execution. The mode to execute this assignment is called exclusive threaded (et). The execution of such an assignment is always done by one special thread, the master thread.

2. An assignment that will be executed by exactly one thread and other threads may be active during its execution. This assignment is executed in a mode called single threaded (st). The execution of such an assignment is always done by one special thread, again the master thread.

3. An assignment that will neither be executed in et-mode nor in st-mode may be executed by more than one thread. This execution can be done by all worker threads, if it is a computation in parallel. Otherwise, in replication, the computation is done by all threads, the master thread and the worker threads. The corresponding execution mode is called multi threaded (mt).

Listing 5.2 shows some SAC-code with annotated execution modes. Its while-loops to compute A and B will be executed in parallel. Thus, the memory allocation has to be done by the master thread. Afterwards, the worker threads will be granted access to the base addresses of A and B to write the results of
the with-loops. Therefore, the allocations are tagged for st-mode. The I/O operation \texttt{Print}(...) also leads to single-threaded execution. Due to the fact that \texttt{SomeExternalFunction}(...) is an imported function from an external library, it has to be executed in et-mode. Figure 5.3 sketches out the execution of this code.

It does not always make sense to replicate code, e.g. an assignment in-between two et-assigns should not be replicated. Such a replication would result in a time-consuming start and termination of the worker threads. Which execution mode such an assignment gets, is shown later in Section 6.2.

### 5.3 Rearranging code

One of SAC's language design characteristics is the absence of side-effects, since it fulfils the Church-Rosser-Property \cite{Ros84}. Due to the absence of side-effects, it is possible to rearrange the code of a SAC-program. This property can be used to decrease the runtime of a program by reducing the number of synchronisations.

As shown in Figure 5.3, each assignment executed in et-mode (short: \textit{et-assign}) causes a full synchronisation of all threads. The rearranging of the code tries to arrange the et-assigns in a way to minimise the number of transitions from et-mode to mt-mode and from mt-mode to et-mode. The transitions concerning et-mode and st-mode are not involved to this topic because this transitions do not cause synchronisations.

The effect of reducing synchronisation is shown in Listings 5.3/5.4 and Figures 5.4/5.5, respectively. Before rearranging the execution toggles between multi-threaded and single-threaded (Listing 5.3 and Figure 5.4). Afterwards, one pair of thread creation and termination is obsolete. Thus, the count of synchronisations is reduced (Listing 5.4 and Figure 5.5).

There is still another benefit of rearranging the code. The memory for each array computed by multiple threads must be allocated by the master thread. This must be performed in st-mode, as mentioned above. Due to the implicit memory management of SAC all memory operations are inserted by the compiler. Up to now, in the existing execution model, the allocation operations for with-loop results are always located adjacent to the corresponding with-loop (shown in Listing 5.5 and Figure 5.6). Thus, although the memory allocations of A and B do not avoid a multi-threaded execution in parallel, this execution would not happen. The assignments that come into question, the with-loops to compute A and B, respectively, have to wait for the occurred
Listing 5.2: Code example with annotated execution modes.

```plaintext
mt ...  
mt a = (b + c) * (d - 1);  
st A = AllocateMemory(...);  
mt A = with ([0, 0] <= [i, j] < [1000, 1000]) genarray([1000, 1000], i * (a - j));  
mt a = a + 2;  
st Print("Hello World");  
st B = AllocateMemory(...);  
mt B = with ([0, 0] <= [i, j] < [1000, 1000]) genarray([1000, 1000], i + j + a);  
et c = SomeExternalFunction(A);  
mt f = fibonacci(x);  
mt ...  
```

Figure 5.3: Schematical code execution of the code of Listing 5.2.
5.3. REARRANGING CODE

<table>
<thead>
<tr>
<th>mt</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>et</td>
<td>c = SomeExternalFunction(e);</td>
</tr>
<tr>
<td>mt</td>
<td>A = with ([0, 0] &lt;= [i, j] &lt; [1000, 1000])</td>
</tr>
<tr>
<td></td>
<td>genarray([1000, 1000], i * (a - j));</td>
</tr>
<tr>
<td>et</td>
<td>d = SomeExternalFunction(f);</td>
</tr>
<tr>
<td>mt</td>
<td>B = with ([0, 0] &lt;= [i, j] &lt; [1000, 1000])</td>
</tr>
<tr>
<td></td>
<td>genarray([1000, 1000], i + j + a);</td>
</tr>
<tr>
<td>mt</td>
<td>...</td>
</tr>
</tbody>
</table>

Listing 5.3: Code arrangement that causes several synchronisations.

<table>
<thead>
<tr>
<th>mt</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>et</td>
<td>c = SomeExternalFunction(e);</td>
</tr>
<tr>
<td>et</td>
<td>d = SomeExternalFunction(f);</td>
</tr>
<tr>
<td>mt</td>
<td>A = with ([0, 0] &lt;= [i, j] &lt; [1000, 1000])</td>
</tr>
<tr>
<td></td>
<td>genarray([1000, 1000], i * (a - j));</td>
</tr>
<tr>
<td>mt</td>
<td>B = with ([0, 0] &lt;= [i, j] &lt; [1000, 1000])</td>
</tr>
<tr>
<td></td>
<td>genarray([1000, 1000], i + j + a);</td>
</tr>
<tr>
<td>mt</td>
<td>...</td>
</tr>
</tbody>
</table>

Listing 5.4: Rearranged code of Listing 5.3 that causes less synchronisations than before.
Figure 5.4: The schematical code execution of the code of Listing 5.3 shows a synchronisation.

Figure 5.5: The schematical code execution of the rearranged code (Listing 5.4) shows no more synchronisation.
5.4. COMMUNICATION STRUCTURE

memory allocation of A and B (the communication mechanism is described in Section 5.4).

A presumably superior sequence to execute this code is shown in Listing 5.6. Now, the memory is allocated before performing the replicated code. This rearranging gives the master thread the opportunity to allocate the memory for A and B before performing the replicated computation of f. Thus, the worker threads' time to wait for the memory of A is decreased by the time to compute f, as illustrated in Figure 5.7.

The usage of rearrangement shows its potential to increase the performance of multi-threaded execution. The used techniques to perform the rearrangement are presented later in Section 6.2. After rearranging the code the individual assignments of the same execution mode are put together in a meta structure by the compiler. This structures are called execution cells. Corresponding to the execution modes three kinds of execution cells exist. Thus, a change of the execution mode calls for a new execution cell. Figure 5.8 shows four execution cells, two st-cells and two et-cells. Communication between the cells happens in terms of data dependencies, as shown in the next section.

5.4 Communication structure

The described new model to execute SAC in parallel needs a communication structure different from the existing one. This structure must support the concurrent execution of st- and mt-cells. Doing so, it must guarantee the completion of a computation before reading its result. The allocation of shared memory must be performed before using the memory, like in Figure 5.6, where A must not be computed before A's memory has been allocated.

In addition to this topic, another problem of the existing communication structure should also be solved. In the existing execution model every data dependency between two concurrently executed with-loops leads to a complete stopping and starting of the whole parallel execution, although this may not be necessary.

Listing 5.7 shows three with-loops that can not be executed within the same parallel section. The computation of A must be finished before the computation of C can start. Using the existing communication structure would yield a full synchronisation. There is only a choice between executing the first pair of with-loops together in a parallel section or the second pair.

Imagine merging the execution of the first pair of with-loops together. This yields a full synchronisation before the computation of C. All threads have to
Listing 5.5: Code arrangement which is impractical for multi-threaded execution due to the adjacencies of memory allocation and usage.

Listing 5.6: Rearranged code of Listing 5.5 that is more practical for multi-threaded execution.
5.4. COMMUNICATION STRUCTURE

Figure 5.6: The schematic code execution of the code of Listing 5.5 shows short distances between memory allocation and usage.

Figure 5.7: The schematic code execution of the rearranged code (Listing 5.6) shows less time to wait before using memory than beforehand in Figure 5.6.
\[
\begin{align*}
\text{a} &= \text{GetInput(...)}; \\
\text{b} &= \text{GetInput(...)}; \\
\text{c} &= \text{SomeExternalFunction(a)}; \\
\text{d} &= \text{SomeExternalFunction(b)}; \\
\text{e} &= \text{GetMoreInput(...c,d,...)}; \\
\text{f} &= \text{SomeExternalFunction(e)};
\end{align*}
\]

Figure 5.8: SAC code with marked execution cells.

\begin{verbatim}
A = with ([0, 0] <= [i, j] < [1000, 1000])
genarray ([1000,1000], i * j);
B = with ([0, 0] <= [i, j] < [1000, 1000])
genarray ([1000,1000], i + j);
C = with ([0, 0] <= [i, j] < [1000, 1000])
genarray ([1000,1000], A[i,j] + i + j);
\end{verbatim}

Listing 5.7: Three with-loops that can not be executed without a synchronisation in between within the existing execution model.
wait until the computation of B is finished before going on with the computation of C. This is done although B is needless to compute C and the computation of A could have been finished a long time ago (as shown in Figure 5.9, left hand side).

**Figure 5.9:** Comparison between strict barrier synchronisation (left hand side) and the new split-phase synchronisation (right hand side) for the example of Listing 5.8.

The problem is located right in the fact, that the existing execution model synchronises always whole blocks of code, the SPMD sections. Each block encapsulates one or more data structures to process. Thus, every synchronisation synchronises all these data structures altogether. Actually, one should synchronise on every single data structure separately.

Therefore, *wait barriers* are added by the compiler. Wait barriers are a kind of “light-weight” barriers, which means that they guarantee a synchronisation, but they may not lead to a stop of all threads. Hence, they can be lifted by
a signal. Wait barriers care for not continuing the parallel execution of the fastest threads until all threads have reached a defined point. This kind of synchronisation, waiting for an event somewhere and signalling it elsewhere, is a so-called *split-phase synchronisation* [CDG+93] (as illustrated in Listing 5.8 as \( \text{signal}(A) \) and \( \text{wait}(A) \)).

Figure 5.9 shows on its right hand side an example for the use of a wait barrier. As long as the computation of \( A \) is not finished and the signal to lift the wait barrier is not sent, no thread is allowed to pass it. Finishing \( A \) lifts the barrier. Now, a seamless transition between the computation of \( B \) and that of \( C \) is possible. This enables the compiler to put all the three \text{with}-loops in one \text{mt}-cell.

This new opportunity to lift barriers before they were reached could lead to a shorter runtime of the program. The comparison of the left hand side with the right hand side of Figure 5.9 gives a small impression of this potential. Nevertheless, all data dependencies are considered and the correctness of the transformation can still be guaranteed.

Merging the second pair of \text{with}-loops of Listing 5.7 together would lead to similar results and implications. Therefore, this is not shown explicitly here.

Besides its usage for inner \text{mt}-cell communication the described split-phase synchronisation is also used to perform the communication between \text{st}-cell and \text{mt}-cells. Synchronisations like waiting for the allocation of shared memory (Figure 5.6) or waiting for a result of a computation (as I/O operations \text{Print}(...) within Figure 5.10) are also possible now.
Figure 5.10: The data dependencies between the mt-cell and the st-cell are performed by split-phase communication as well.
Chapter 6

Realisation of the new execution model

This chapter shows techniques for the realisation of the new execution model. After handling the execution modes in Section 6.1, it continues with describing the rearranging algorithm (Section 6.2). The realisation of the split-phase communication is presented in Section 6.3 before having a closer look on implications on the dynamic memory management system in Section 6.4.

6.1 Execution modes

6.1.1 Any threaded execution mode

As mentioned in Section 5.2, each assignment is executed in one of three execution modes. Due to their restrictions some assignments must be executed in a specific execution mode. The execution mode of all other assignments can be freely chosen between et-, st- or mt-mode. Their execution mode will be determined later on.

As long as the final mode is not determined, these assignments are tagged to be in any threaded execution mode (at-mode). The at-mode is a temporary attribute, only used as a placeholder for the actual execution mode that is determined later on. Thus, at-mode just says the assignment may be replicated, but this is not known now. This flexibility will be restricted by handling the functions (Section 6.1.2). If an assignment is still in at-mode afterwards, its execution mode will be determined by the rearrangement, as shown in Section 6.2.
6.1.2 Handling functions

The introduction of execution modes does not consider the question how to handle function calls. The execution mode of a function $f$ is determined by its assignments, defined as $ass(f)$. The execution mode of an assignment $a$ is returned by $exmode(a)$. Then, the execution mode of a function $f$ is defined as:

$$
exmode(f) := \begin{cases} 
  et \quad & \exists a \in ass(f) : exmode(a) = et \\
  et \quad & \exists a, b \in ass(f) : exmode(a) = st \land exmode(b) = mt \\
  st \quad & \forall a \in ass(f) : exmode(a) \in \{at, st\} \\
  mt \quad & \forall a \in ass(f) : exmode(a) \in \{at, mt\} \\
  at \quad & \text{otherwise}
\end{cases}
$$

The application of a function $f$ gets the execution mode of the function. Hence, a change of $f$'s execution mode causes a change of all its applications. This may cause an execution mode change of the function calling $f$ and so on.

Therefore, the determination of the function’s execution mode must take place in a fix point iteration. The iteration terminates if no more changes of execution modes are performed. It terminates always because in each iteration the execution mode of a function is either unchanged or changed to a mode the function have not had any time before.

Thus, the execution mode of a function is something like the least common denominator of its assignments’ execution modes. This guarantees a consistent execution of the associated SAC program.

The representation of a SAC program in the compiler is in *Single Static Assignment* form (SSA) [RWZ88]. The SSA form avoids complex expressions and represents loops and conditionals as special functions. Hence, the new multi-threaded execution model handles loops and conditionals as functions. Each conditional function includes a remaining conditional. The handling of this last conditional is identically to the handling of a function and can be adopted one-to-one.

After performing the computation of the functions’ execution modes, each function still in at-mode is specialised into versions of et-, st- and mt-mode, respectively.
6.2 Rearranging assignments

6.2.1 Basics of graph theory

The rearranging algorithm is based on graph theory. Therefore, some short basic definitions are given, adapted to the context in use (see [Tei02] for more details).

- $G = (N, E, \varphi)$ is a graph :=
  1. $N$ is a set whose elements are called nodes,
  2. $E$ is a set whose elements are called edges,
  3. $\varphi : E \rightarrow N'$ is the function of incidence,
     $N' = (N \times N) \setminus \{(n_i, n_i)\}, 1 \leq i \leq |N|$.

- $P, Q \in N$ are adjacent := $\exists e \in E : \varphi(e) = (P, Q)$

- $P \in N$; out degree $V^+(P : G) :=$ number of edges, starting in $a$,
  $P$ is a sink := $V^+(P : G) = 0$

- Graph $G' = (N', E', \varphi')$ is a subgraph of $G = (N, E, \varphi)$ :=
  $N' \subseteq N \land E' \subseteq E \land \varphi = \varphi'$

- Graph $G' = (N', E', \varphi')$ is called the subgraph created by $N' \subseteq N$ of $G = (N, E, \varphi)$ :=
  $E' = \{(P, Q) \in E : P, Q \in N'\} \land \varphi' = \varphi'_{|E'}$

- Let $f$ be a function in SAC without dead code\(^1\). Let $A = a_1, \ldots, a_n$ be the assignments of $f$ (a conditional of $f$ is regarded as an assignment).
  Let $N$ be a set of nodes, $|N| = n$. Let a bijective mapping exist between $A$ and $N$ ($n_1 \mapsto a_1, \ldots, n_n \mapsto a_n$).
  The data dependency $\text{dep}_f$ is defined as $\text{dep}(n_i, n_j, f) :=$
  $$\begin{cases} 
  \text{true} & \text{if } a_j \text{ is data dependent of } a_i \text{ in } f \\
  \text{false} & \text{otherwise}
  \end{cases}$$
  $G(N, E, \varphi, f)$ is called the data flow graph of $f :=$
  $$\forall (n_i, n_j) \in E : \text{dep}(n_i, n_j, f) = \text{true} \land \forall n_i \in N \text{ each data dependency of } a_i \text{ in } f \text{ is mentioned in } \varphi$$

Each element of $N$ gets an attribute (colour) $c(N) \in \{a, e, s, m\}$, for any, exclusive, single or multi threaded execution mode of the corresponding

\(^1\)The restriction concerning the dead code facilitates the presentation of the algorithm. The algorithm itself may well handle dead code.
assignment in \( f \). Figure 6.1 shows an example of a function and its representation via the associated data flow graph.

A specific code listing of a SAC program is one of probably many possible semantical equivalence arrangements of the program. The listing is only one result of a traversal of the program’s data flow graph without violating the existing data dependencies. The rearranging algorithm to be presented aims at generating a code traversal out of the data flow graph, which is preferably well-suited for concurrent execution.

As a technical extension of the execution model described in Chapter 5, the master thread is a kind of extended worker thread. In addition to its st-mode master work it executes the mt-mode worker code as well. Thus, the processor performing the master thread is as fully loaded as the worker threads’ processor.

### 6.2.2 The rearranging algorithm

The rearranging algorithm works on function level. It is divided into three consecutive steps.

1. Each changing from mt-mode into et-mode results in a time consuming synchronisation, whereas each changing from et-mode into mt-mode causes the setup or restart of the multi-threaded execution environment. The first step aims at reducing the number of these changes. It takes the data flow graph of a function and presorts the function’s assignments by grouping them into sets. Each assignment of a set has an identical execution mode and is data independent of all other set members. The
first step returns a list of sets at which the assignments of the first set have to be executed before the assignments of the second set and so on.

2. The second step takes this list of sets and returns a list of lists of sets. Each list of sets corresponds to a set in the first step. Each set of within a list contains assignments $A$ with identical distances to sets of another list which contain assignments $A$ depend on.

3. At the end, the third step takes these list of list of sets of assignments and performs a final arrangement of the assignments. Thus, this step returns a list of assignments, bottom-up builted. The position of an assignment depends on the explicit arrangement of data depended assignments backwards in the resulted list.

Figure 6.2 shows an example of a data flow graph. In the following this graph is used to illustrate the procedure of the algorithm.
6.2.3 Step 1 - Reducing execution modes changes

The first step of the rearranging algorithm is shown in Figure 6.3. To reduce the count of execution mode changes between et- and mt-mode the algorithm must find clusters of assignments (nodes) in et-mode and mt-mode, respectively, as large as possible. As mentioned in Section 6.1, each function containing an assignment in et-mode, is tagged as et-mode as well.

Thus, the execution mode of the function’s return statement is in et-mode. Therefore, the algorithm starts putting all et-coloured sinks of the function’s data flow graph \( G \) into a set\(^2\). This set will be the last set of the list returned by the first step. Afterwards, the algorithm takes the subgraph \( G' \), created by \( G \)'s nodes without the nodes already collected, and puts all et-coloured sinks of \( G' \) into a new set. This set will be the new head of the resulting list. This procedure continues until \( G' \) includes no more et-coloured nodes.

The st- or at-property of an assignment gives the opportunity to execute it even in et-mode. Therefore, the algorithm goes on collecting at- and st-coloured nodes, at which every of this at- or st-collecting pass is followed by et-collecting passes. This collecting of nodes continues until no more et-, at- or st-coloured sinks in \( G' \) are available.

Now, the algorithm switches into collecting mt-coloured nodes (position 3a in Figure 6.3). Therefore, the st-coloured nodes are the last nodes collected before because assignments in st-mode can be executed in parallel to assignments in mt-mode. In contrast to the collecting of et-coloured nodes, at which dependent nodes are put into consecutive sets, dependent mt-coloured nodes are put into sets with sets of at-coloured and st-coloured nodes in-between if possible. For short, the algorithm puts as few other sets as possible in between “et-sets”, but as many other sets as possible in between “mt-sets”.

This procedure is motivated by reducing the idle time of threads, waiting at a wait barrier. The reason to handle at-coloured nodes earlier than st-coloured nodes (as shown in Figure 6.3 on position 3b) is to increase the distance between assignments in st-mode and dependent assignments in mt-mode. Thus, the master thread is given as much time as possible to perform its st-cell before the first worker thread reaches a dependent mt-cell.

Due to the use of the SSA form, a constructed data flow graph does not contain cycles. Therefore, the algorithm always finds a sink in the data flow graph \( G \) or any of its subgraphs. Hence, the algorithm always terminates. Performing a data flow graph without et-mode nodes is supported as well, but then, this first step cannot reduce the number of changes between et-mode and mt-mode.\(^3\)

\(^2\)Due to the absence of dead code the amount of the set cannot be larger than 1.
Given the coloured data flow graph \( G = (N, E, \varphi, f) \), the algorithm to reduce the number of execution mode changes in \( f \) is as follows:

1. \( S := \text{nil}; \ S_{hlp} := \emptyset; \ G'(N', E', \varphi') := G; \)

2. (a) \( S_{hlp} := \{ n \in N' : V^+(n : G') = 0 \land c(n) = e \}; \)
   \( N' := N' \setminus S_{hlp}; \)
   \( G' := \text{subgraph created by } N' \)
   \( \text{If } (S_{hlp} \neq \emptyset) \{ S := S_{hlp} : S; S_{hlp} := \emptyset; \text{goto 2a}; \} \)

   (b) \( S_{hlp} := \{ n \in N' : V^+(n : G') = 0 \land c(n) = a \}; \)
   \( N' := N' \setminus S_{hlp}; \)
   \( G' := \text{subgraph created by } N' \)
   \( \text{If } (S_{hlp} \neq \emptyset) \{ S := S_{hlp} : S; S_{hlp} := \emptyset; \text{goto 2a}; \} \)

3. (a) \( S_{hlp} := \{ n \in N' : V^+(n : G') = 0 \land c(n) = m \}; \)
   \( N' := N' \setminus S_{hlp}; \)
   \( G' := \text{subgraph created by } N' \)
   \( \text{If } (S_{hlp} \neq \emptyset) \{ S := S_{hlp} : S; S_{hlp} := \emptyset; \text{goto 4}; \} \)
   \( \text{else goto 4;} \)

   (b) \( S_{hlp} := \{ n \in N' : V^+(n : G') = 0 \land c(n) = a \}; \)
   \( N' := N' \setminus S_{hlp}; \)
   \( G' := \text{subgraph created by } N' \)
   \( \text{If } (S_{hlp} \neq \emptyset) \{ S := S_{hlp} : S; S_{hlp} := \emptyset; \text{goto 3b}; \} \)

   (c) \( S_{hlp} := \{ n \in N' : V^+(n : G') = 0 \land c(n) = s \}; \)
   \( N' := N' \setminus S_{hlp}; \)
   \( G' := \text{subgraph created by } N' \)
   \( \text{If } (S_{hlp} \neq \emptyset) \{ S := S_{hlp} : S; S_{hlp} := \emptyset; \text{goto 3b}; \} \)
   \( \text{else goto 3a;} \)

4. If \( N' \neq \emptyset \) goto 2a;

5. Return \( (S); \)

**Figure 6.3:** The first step of the rearranging algorithm. This step reduces the execution mode changes between et-mode and mt-mode assignments.
$S_1 = \{A\}$, $S_2 = \{B\}$, $S_3 = \{C\}$, $S_4 = \{D\}$, $S_5 = \{E, F, G\}$, $S_6 = \{H\}$, $S_7 = \{I\}$, $S_8 = \{J, K, L\}$, $S_9 = \{M\}$, $S_{10} = \{N, O\}$, $S_{11} = \{P\}$, $S_{12} = \{Q\}$

**Figure 6.4:** The data flow graph of Figure 6.2 after the first step of the algorithm. The resulting sets of nodes are marked.

The first step of the algorithm builds several sets of nodes with identical execution mode, as illustrated in Figure 6.4. The resulting list $S = [S_1, \ldots, S_{|S|}]$ shows the order of the function’s execution. After performing the first step, each set contains data independent nodes of the same colour (execution mode). These sets are the initial points for the second step of the rearranging algorithm.

### 6.2.4 Step 2 - Preorder nodes of the same set

The next step of the rearranging algorithm is shown in Figure 6.5. This step of the algorithm does not change the order of the sets $S_i$. This order is fixed. Now, an order within the individual sets is created, the sets $S_i$ are transformed into corresponding lists $L_i$. Such a list $L_i$ contains sets $l_{i,j}$, $1 \leq j \leq |S_i|$. As
Let the list of sets \( S = [S_1, \ldots, S_m] \); \( S_i = \{n_{i1}, \ldots, n_{i|S_i|}\} \) be given. Let the distance \( \text{dist} \) of a node \( n_{ij} \) concerning the function \( f \) of a data flow graph \( G = (N, E, \varphi, f) \) be defined as
\[
\text{dist}(n_{ij}, f, S) := \begin{cases} 
\min\{n_{a,b} \in S_a : \text{dep}(n_{a,b}, n_{i,j}, f) = \text{true} \} : & \text{if such an } a \text{ exists} \\
|S| : & \text{otherwise}
\end{cases}
\]
Thus, \( 1 \leq \text{dist}(n_{ij}, f, S) \leq m \) for all \( n_{ij} \). The algorithm to presort the elements of the sets \( S_i \) into lists \( L_i \) is as follows:

- \( L_a := \text{nil}, (1 \leq a \leq m); \) \( S_{hlp} := \emptyset; \) \( i := 1; \) \( j := 0; \) \( d := 0; \)
- 1. \( d := 1; \)
  2. (a) \( j := 0; \)
     (b) i. If \( (\text{dist}(n_{ij}, f, S) = d) \) \( \{S_{hlp} := S_{hlp} \cup n_{i,j}\}; \)
     ii. \( j := j + 1; \)
     iii. If \( (j \leq |S_i|) \) goto 2(b)i;
     (c) If \( (S_{hlp} \neq \emptyset) \) \( \{L_i := S_{hlp} :: L_i; \) \( S_{hlp} := \emptyset;\}\}
     (d) \( d := d + 1; \)
     (e) If \( (d \leq m) \) goto 2a;
  3. \( i := i + 1; \)
  4. If \( (i \leq m) \) goto 1;
- Return \( (L_1, \ldots, L_m) \);

**Figure 6.5:** The second step of the rearranging algorithm. This step presorts the elements of the sets, given by the first step of the algorithm (Figure 6.3).
\[L_1 = \{A\}, L_2 = \{B\}, L_3 = \{C\}, L_4 = \{D\},
L_5 = \{E, F, G\}, (\text{dist}(E, f, S) = \text{dist}(F, f, S) = \text{dist}(G, f, S) = 11)
L_6 = \{H\}, L_7 = \{I\},
L_8 = \{J, L\}, \{K\}, (\text{dist}(J, f, S) = \text{dist}(L, f, S) = 3; \text{dist}(K, f, S) = 1)
L_9 = \{M\},
L_{10} = \{N, O\}, (\text{dist}(N, f, S) = \text{dist}(O, f, S) = 2)
L_{11} = \{P\}, L_{12} = \{Q\}\]

Figure 6.6: The result of the second algorithm step, using the output of the first step (Figure 6.4).

the result of the second step those sets \(l_{i,j}\) contain the nodes of the data flow graph.

Each node \(n\) gets an attribute, called \textit{distance}. The distance shows the minimum of how many sets \(S_i\) are between \(n\)'s set and each of the nodes, \(n\) depends on. As an example, in Figure 6.6 the distance of \(K\) with respect to the data flow graph (\(\text{dist}(K, f, S)\)) is 1 (\(K \in S_8\) depends on \(I \in S_7\)). Compared to this \(\text{dist}(J, f, S)\) is 3 (\(J \in S_8\) depends on \(F \in S_5\)).

According to this attribute, the nodes of a set \(S_i\) are assigned to the sets \(l_{i,s} \in L_i\) in ascending order with respect to their distances. If the distances of two nodes are equal, they will be assigned to the same set. The reason to order the assignments in this way is to avoid executing dependent assignments one after another directly.

It is superfluous to perform this second step of the algorithm within sets \(S_i\) of nodes that are coloured "et". It is also nonsensically to have an assignment's distance larger than the distance to the previous assignment in et-mode. This facts are ignored to keep things simple.

Figure 6.6 shows the example mentioned before after performing this second step of the algorithm. The interesting lists are \(L_5, L_{10}\) and \(L_8\) in particular. Due to the fact that \(K\) depends on \(I\) in \(L_7\), it is arranged behind \(J\) and \(L\), that depends on nodes of \(L_5\).

6.2.5 Step 3 - Fixing the order

The last step of the algorithm fixes the order of the assignments. The result of the second step produces an almost defined execution order. Only the arranging of members of the same set \(l_{i,j}\) still needs to be performed.

The algorithm performs bottom-up to produce the final list of nodes, starting with an empty list \(\mathcal{N}\). If \(l_{i,j}\) contains only one member, this node will be
6.2. REARRANGING ASSIGNMENTS

placed at the top of \( NL \) to create the new \( NL \). Otherwise, the algorithm will choose the element of \( l_{i,j} \) that has a data dependent node in \( NL \) as far away as possible. It removes the element from the \( l_{i,j} \), adds it to the top of \( NL \) and continues with \( l_{i,j} \). If \( l_{i,j} \) is empty, the algorithm takes the set before \( l_{i,j} \). This is done for all sets. Figure 6.7 shows the full third step of the algorithm.

The motivation to perform the third step bottom up is the structure of most SAC programs:

1. The sets at the beginning of a function are usually bigger than the sets at its end. The data dependencies increase in the course of the function. Therefore, the number of possible execution orders is restrained more and more.

2. The allocation of shared memory is always done before performing computation in parallel. It is usually independent of the parallel computation. This leads to a big set at the beginning of a function. Choosing the execution order of the allocations by random would restrain the possible orders of all other assignments much more than performing bottom-up.

The result of the third step of the accompanying example is presented in Figure 6.8.

The interesting point is the order of the nodes \( J, L, N, O \). The algorithm avoids the inapt orders \([\ldots, L, J, \ldots, N, O \ldots]\) and \([\ldots, J, L, \ldots, O, N \ldots]\). These orders are inapt because \( N \) depends on \( J \) and \( O \) depends on \( L \), respectively. Thus, if \( N \) and \( J \) are only separated by \( M \), the probability of \( J \) being finished by the slowest thread while the fastest thread does not idle at the wait barrier before \( N \) is very low. The order \([\ldots, J, L, M, N, \ldots]\) raises this idle time, because the fastest thread can perform assignment \( L \) in-between. Another well suited order would be \([\ldots, L, J, M, O, \ldots]\).

6.2.6 Building execution cells

The rearranging algorithm fixes the execution order. Afterwards, the assignments have to be combined within execution cells to satisfy the execution model described in Section 5.3. This makes the decision, whether an at-mode statement must be replicated or not.

Performing bottom-up, reaching an at-mode statement causes a changing of its execution mode into the execution mode of the last statement passed. As pointed out in Section 5.2, each function and with it its return statement owns
Let the list of lists \( L = [L_1, \ldots, L_m] \); \( L_i = [l_{i,1}, \ldots, l_{i,|L|}] \);
\( l_{i,j} = \{n_{i,j,1}, \ldots, n_{i,j,|L|}\} \) be given. Let \( G = (N, E, \varphi, f) \) be the concerning data flow graph. Let the execution distance \( edist \) of a node \( n_{i,j,k} \) concerning a list \( A \) of nodes be defined as
\[
edist(n_{i,j,k}, f, A) := \begin{cases} 
\min_{l \in A} \text{dep}(n_{i,j,k}, f, l) & \text{if such an } l \text{ exists} \\
|N| & \text{otherwise}
\end{cases}
\]
The algorithm to complete the rearranging returns a list \( NL \) of nodes is defined as follows:

- \( NL := nil; i := m; \)
- 1. \( j := |L|; \)
  2. (a) \( k := |l_{i,j}|; \)
     (b) \( tmp := n_{i,j,k}; \)
     (c) i. If \( edist(n_{i,j,k}, f, NL) > edist(tmp, f, NL) \) \( tmp := n_{i,j,k}; \)
     ii. \( k := k - 1; \)
     iii. If \( (k \geq 1) \) goto 2(c)i;
  (d) \( NL := tmp :: NL; \)
  (e) \( l_{i,j} := l_{i,j} \setminus tmp; \)
  (f) If \( (l_{i,j} \neq \emptyset) \) goto 2a;
  (g) \( j := j - 1; \)
  (h) If \( (j \geq 1) \) goto 2a;
- 3. \( i := i - 1; \)
- 4. If \( (i \geq 1) \) goto 1;
- Return \( (NL) \);

**Figure 6.7:** The third step of the rearranging algorithm. This step finishes the rearranging getting the output of the algorithm’s second step (Figure 6.5).

\[
NL = [A, B, C, D, E, F, G, \ldots] \quad (edist(G, f, NL) = 4, edist(F, f, NL) = 3, edist(E, f, NL) = 1)
\]
\[
\ldots H, I, J, L, \ldots \quad (edist(L, f, NL) = 4, edist(J, f, NL) = 3)
\]
\[
\ldots K, M, N, O, \ldots \quad (the \ order \ of \ N \ and \ O \ is \ chosen \ by \ random)
\]
\[
\ldots P, Q]
\]

**Figure 6.8:** The result of the last algorithm step, using the output of the second step (Figure 6.6).
a function mode unequal “at”. Thus, there is always a last statement with execution mode different to “at”. If the performed at-mode statement is a function application, the binding to the concerning specialised function is set.

At last, the execution cells are created via collection of all consecutive assignments of the same execution mode. Each collection is transformed into an execution cell, corresponding to their execution modes.

To show the building of these execution cells, Figure 6.9 finishes the series of examples illustrating the rearranging algorithm. The order of execution is $st_1 | mt_2, et_3, st_4 | mt_5, et_6$.

### 6.3 Split-Phase Communication

The fork/join model and with it the existing multi-thread model for SAC uses its fork and join statements for two different purposes, firstly, to identify the
borders of the concurrently executed regions and secondly, to label areas without any data dependency between concurrently evaluated expressions. These areas are identical with the concurrently executed regions. These accordances do not exist in the new execution model because of the wait barriers within the concurrently executed regions. Hence, a new mechanism for synchronisation must be developed.

Due to their usage, two different kinds of wait-barriers can be identified. They differ in the kind of action that is synchronised and in the way of signalling the lifting of the barriers.

1. Waiting for the completion of data objects’ computation is used every time the data object is not computed by the waiting thread alone. This can appear

   (a) as a one-to-many communication (the master thread computes something, the worker threads need and can/do not compute it themselves),

   (b) many-to-one communication (the worker threads compute something altogether that the master thread needs) or

   (c) as a many-to-many communication (the worker threads compute something altogether that the worker threads need).

2. Waiting for the memory allocation of a concurrently evaluated expression. The allocation is always done by the master thread and the memory will be filled by the worker threads afterwards. This is always a one-to-many communication. The difference between the one-to-many communication described beforehand is technical. The information of being evaluated or not can be added to the variable itself, because the variable already exists. In contrast, waiting for the allocation of a variable needs a special buffer, existing before and independently of the depending variable.

The following two subsections describe, how this two subtypes of communication are realised.

### 6.3.1 Synchronisation of data dependencies

A complete communication structure to support all kind of communication, like one-to-one communication in-between all threads or broadcasts by each thread, would produce a lot of overhead.
6.3. **SPLIT-PHASE COMMUNICATION**

The communication needed here is very restrained. One-to-one communication is not needed at all. One-to-many communication and many-to-one communication are only used with the master thread as the root of communication. Hence, a complete communication structure to support all kinds of communication is superfluous here.

Each worker thread must process a (maybe empty) part of a concurrent with-loop. This part is determined by the scheduling of the with-loop [End02]. To synchronise within the new execution model each parallel computed array gets an extra synchronisation structure to support the communication requirements, an *array of flags*. Each flag corresponds to one of this parts, i.e. one flag per thread.

The array of flags is created by the master thread and made available to the worker threads in combination with the memory allocation of the actual data array. All flags are unset for initialisation. After computing its part of a parallelised with-loop \( w \), each thread sets its flag. Afterwards, the thread continues with subsequent computations. If it reaches the barrier, to wait for the computation of \( w \), the thread waits until all flags of \( w \)’s array of flags are set. The “signal” to lift the barrier is mapped implicitly to this inspection. If all threads finished their parts of \( w \)’s computation, all flags will be set and also the wait barrier is lifted.

As an example, Figure 6.10 shows the computation of \( A \). The computation is scheduled for four threads: \( T_0, \ldots, T_3 \). Corresponding to the number of threads, \( A \)’s array of flags has four entries. The \( T_0, T_2 \) and \( T_3 \) have completed their parts \( (P_0, P_2, P_3) \) of \( A \), \( T_1 \) is still on its way. Thread \( T_3 \), waiting at the barrier, inspects the flags until all flags are set.

This procedure must be extended to use it with the results of fold-with-loops computed in parallel. In the existing concurrent execution model the master thread allocates an array before starting the concurrent execution. This array has one entry per thread to hold its partial result. After finishing its computation and writing its partial result to this global array, the threads start computing the global result via a tree-like structure, as illustrated in Figure 4.3. In the end the master thread and only the master thread gets the complete result. Thus, if the worker threads need the global result, the master thread must broadcast it. A characteristic of the existing model is the existence of this one array to be reused for all fold-with-loops.

One global structure to hold all partial results is not suitable in the new model. The size of mt-cells may vary, and multiple mt-cells may be executed at the same time. Thus, each fold-with-loop gets a construct of the memory for flags (as describes above) and a buffer to store the partial result of each thread.
Figure 6.10: Example of the wait barrier’s functionality. While the threads $T_0$, $T_2$ and $T_3$ have finished their share of $A$, $T_1$ is still computing. $T_3$ has reached the barrier and waits for the last flag to show that $A$ is available.

Additionally, this construct gets a pointer to the fold function as well. Hence, an extended array of flags (Figure 6.11) must be allocated and its base address must be distributed by the master thread.

Figure 6.11: Extended array of flags to be used with fold-with-loops.

After completing its part, a worker thread must write its result into the corresponding field of the buffer. Then, it sets the flag to signal the completion of its partial result. This sequence of writing the result and signalling it is essential. It avoids the reading of an incomplete result by other threads.
If a thread reaches a wait barrier to wait for the result of a `fold-with-loop`, it must act differently as well. After inspecting the flags and before continuing execution, it must compute the over-all result of the `fold-with-loop` by itself, using the corresponding fold function. Each thread behaves in the same way.

### 6.3.2 Waiting for the allocation of shared memory

It is impossible to use the synchronisation technique presented in Section 6.3.1 to let the worker threads wait for shared memory allocated by the master thread. It is impossible because the base address of the variable's memory is not known before the memory is allocated, and with it the address of the array of flags. Thus, the worker threads need another location to check whether the memory has been allocated or not. Afterwards, the worker threads must be able to get the descriptor and the base address of the new variable to have all information they need.

The realisation is inspired by that of the existing execution model, i.e. via a separate global buffer. The address of this buffer is broadcast to the worker threads at their creation. The buffer contains a $3 \times n$ array. Each column contains the information for one array, ordered consecutive to their appearance in execution. The first row contains one flag per variable, initialised as unset. Rows two and three contain two memory addresses, the base and the descriptor address of the corresponding array, both initialised as NULL. This construct enables the worker threads to test the global buffer for the availability of specific variables. To support more than $n$ variables, a new buffer can be built, e.g. to support recurisons. The new buffer is linked via a special pointer to the old one. Figure 6.12 shows an example of such a buffer. While all allocations of the first array have finished, the allocation of the second array's memory is still in progress.

![Figure 6.12: Global accessible buffer to broadcast memory addresses.](image)
Each mt-cell is executed by each worker thread. The arrangement of the mt-cells is identical for each worker thread. Hence, all worker threads need all shared arrays and need these array in the same order. Therefore, they find each shared array, needed for the first time, always in the next column of the global buffer.

Whenever the master thread allocates memory for an array to distribute it to the worker threads, it first writes the array’s base address and its descriptor into the corresponding fields of the array. Afterwards, it sets the flag to signal the worker threads the completed allocation. This order guarantees that no thread can read a potentially incorrect memory address.

6.4 Consequences for the memory management

As shortly mentioned before, SAC includes an implicit dynamic memory management [GT04], based on reference counting [Col60]. Reference counting is a technique to keep count of how many pointers refer to a given data object. If the counter of a data object decreases to zero, the memory of the object can be de-allocated.

The handling of reference counters in the existing concurrent execution model is simple. The only thread that de-allocates shared memory is the master thread. To guarantee the shared memory to de-allocate is not in use anymore the master thread waits until all worker threads have terminated. This procedure would cause problems, if it was used in the new execution model because it does not provide memory reuse within the areas of concurrent execution. Due to this larger areas, the needed amount of memory would be much higher than before. Thus, the reference counting mechanism must work with shared data in a way to enable de-allocating or reusing memory within concurrent execution.

The first approach to have in mind is straightforward, to share the reference counter as well as the variable. The value of the reference counter equals the number of existing references to the variable in the whole system. To avoid inconsistencies, a mutex lock [And99] must be used to guarantee exclusive access to the reference counter.

This solution would solve the problem, but would also lead to a poor runtime performance. By using a shared object $O$, each thread must work with $O$’s reference counter every time it touches $O$. This would lead to many threads waiting in a queue to grant access to $O$’s reference counter. While the number of threads increases, the length of the queue increases as well, which scales
poorly. To improve the first approach, the reference counter is divided into two layers, a global counter and \( n \) local sub-counters, where \( n \) is the number of threads. Such a \textit{mt-descriptor} is shown in Figure 6.13. A local sub-counter equals the amount of references the depending thread owns of this variable. Although the sub-counter is called local, it is part of the globally known descriptor. The global counter just counts the number of threads, that have at least a reference to the variable. Each time a thread must manipulate the reference counter, it manipulates its local reference counter as long as it is not set to zero. If so the thread must decrease the global thread counter by one. Thus, just the global thread counter must be protected by a mutex lock. Due to the fact of one thread only manipulates this counter one time, the number of attempts to get the lock will just increase linear with the number of threads.

Another thing to keep in mind is the memory hierarchy of modern computer systems. These systems have several levels of memory to gain a good performance. So-called \textit{cache memories} bridge the speed gap between processor registers and main memory [Smi82].

The usage of a reference counter like in Figure 6.13 would cause cache invalidations and reloads due to the adjacent placement of the local counters.
This would lead to the so-called false sharing phenomenon [KLE93] and furthermore to cache thrashing [Gla89]. Everytime a thread changes its local sub-counter, this yields an invalidation of the cache line in every other processor’s cache, where the descriptor and with it the sub-counter are located (to ensure cache coherency). Then, when another thread wants to manipulate its local sub-counter, it must reload the cache line before changing it.

One solution to avoid this effect would be real local reference counters, apart from the global one. This would cause a kind of distributed memory management system, where each thread must allocate a small part of the reference counter by itself. Another solution to avoid false sharing is padding [BGS94]. It is a mechanism to keep the distance in memory between two subcounters long enough to have only one sub-counter per cache line. Such a padded reference counter for a shared variable is shown within the descriptor in Figure 6.14. The shape vector of the variable is replicated to avoid the same effects as well.

Another improvement compared to the descriptor presented before is the way each thread gets access of the descriptor. While in Figure 6.13 each thread gets
the same start address of the descriptor in memory, in Figure 6.14 each thread receives a different address. The memory access of all threads, even the master thread’s, is identical now. Thus, the routines to manipulate the descriptor and de-allocating the memory can be identical for the worker threads and for the master thread. This avoids specialised code and a conditional to test on being the master thread or a worker thread.
Chapter 7

Optimisation opportunities

7.1 Topological synchronisations

Everytime a thread tries to get access to shared data it must wait at a barrier if the computation of the data is not finished. The thread must wait if even the part of the data it would like to get access to is already computed, e.g. by the thread itself, as shown in Listing 7.1. A thread that has finished the computation of \( A \) could continue computing \( B \) if both with-loops were scheduled identically. Each thread would need exactly the those entries of \( A \) to compute its part of \( B \) it has already computed by itself.

\[
\begin{align*}
\ldots \\
A &= \text{with } (lb <= iv < ub) \\
\text{genarray}(size, \ldots iv \ldots) ; \\
signal(A) ; \\
wait(A) ; \\
B &= \text{with } (lb <= iv < ub) \\
\text{genarray}(size, \ldots A[iv] \ldots) ; \\
\ldots
\end{align*}
\]

Listing 7.1: Example to show the potential of partial synchronisation.

The way the synchronisation of an array is performed in the new execution model restricts the synchronisations to whole arrays, as shown in Figure 7.1, left hand side. Due to the fact the threads \( T_1 \) and \( T_4 \) do not know which areas of \( A \) have been computed yet\(^1\), they have to wait for \( A \)'s completion.

\(^1\)Therefore, the computed part is shown as uniform undifferentiated area.
If they knew which areas of $A$ are already available, they would be able to go on processing $B$, because they have computed the parts of $A$ they need by themselves.

**Figure 7.1:** Using standard split-phase synchronisation the threads $T_1$ and $T_4$ have to wait for the complete computation of $A$ before going on to compute $B$ (left hand side). Via the knowledge of the array's scheduling each thread can determine the parts of $A$ it needs to compute $B$. Performing this topological synchronisation enables $T_1$ and $T_4$ to proceed computing $B$ (right hand side).

Restricting synchronisation to areas of $A$, so-called *topological synchronisation* [SM96], solves this problem. To support topological synchronisation, two partial aspects are relevant:

- The areas of an array one thread needs must be determined. The usage of static scheduling schemes enables the compiler to determine these areas while dynamic schedulings could only be analysed at runtime.

- The thread must be able to determine if these areas are ready or not. This can be done by interpreting the array’s array of flags in a more detailed way. As described in Section 6.3.1 a thread has to inspect all flags of this array to perform the synchronisation. By checking only those flags that signals the finished computation of the needed areas, topological synchronisation can be performed.
If it is statically known which data is needed and that a thread computes these data by itself, it will be even possible to remove a signal wait combination completely (as shown in Figure 7.1, right hand side). Each thread processes the area of A it needs to compute B by itself. Therefore, the barrier is removed, and both T\textsubscript{1} and T\textsubscript{4} can continue processing.

The same idea of topological synchronisation can be used the other way round to increase the performance of the memory management. The memory of an array could be rewritten as long as the overwritten data is not needed anymore. This data reuse would improve the current data reuse by the usage of less memory.

7.2 Balanced execution of st-cells

Each st-cell is executed by the same thread, the master thread. Assuming the runtime of mt-cells are identical per thread, the master thread (as worker thread 0) will need more time to perform its tasks than the other threads. To equalise these runtime differences two solutions are possible.

One option to move some workload from the master thread to the worker threads is reducing the master thread’s part of a with-loop performed in parallel by adapting the scheduling accordingly. This assumes a kind of cost model to determine the size of the portion being computed by the master thread. Some more detailed information about scheduling with-loops in SAC can be found in [End02].

Changing the thread that processes an st-cell is another option. This change from having one fixed thread performing st-cells to several threads doing different cells, probably in parallel, will also decrease the master thread’s workload. The determination of the thread to compute an st-cell or even an st-assignment can be done either via a static method at compile time (like a cyclic distribution) or by a dynamic way during runtime (like first come, first served).

Performing assignments in st-mode by different threads leads to several difficulties. The assignments that deal with I/O must be synchronised. The memory management subsystem must be adapted to work with those assignments that handle memory functions. Therefore both, synchronising I/O and adapting the memory manager, require essential changes to the compiler.
7.3 Distributed memory allocation

The current memory management of SAC is well-suited for the existing concurrent execution model. It focuses on the master thread to allocate and de-allocate shared memory. This preference leads to restrictions in the new multi-thread execution model. The allocation and de-allocation of shared data generates several st-cells with all their problems concerning concurrent execution (as pointed out in Section 7.2). Hence, to avoid these restrictions, the memory management system must be adapted to a more efficient execution of the new execution model. Additionally, this adaption could be used to run SAC on distributed memory machines like clusters.

Performing a with-loop in mt-mode, each thread could allocate the memory it needs to store its results by itself. This would reduce the number of st-cells and increase the count of executions being done in parallel. Such a kind of distributed memory allocation, as used in SPLIT-C [CDG+93] or NestStep [Keß00], reduces the idle time of worker threads.

However, the benefit of performance in allocating memory comes along with high costs to access an element of such a distributed array which was not allocated by the thread itself. An additional layer in memory is needed to translate the memory addresses of such an array to a global address space. For example SPLIT-C provides a global address space with a clear concept of locality, but its so-called spread arrays must be declared specially. Thus, distributed memory allocation has only advantages in running SAC programs on distributed memory machines or if the most memory access of a data is done by the thread that has allocated it.

7.4 Parallelisation beyond function borders

The new execution model has an inconvenient restriction. A function which contains both, assignments in st-mode and assignments in mt-mode, must be called by its context in et-mode. Thus, the et-mode applications of this function may enforce a synchronisation, as shown in Figure 7.2. Calling function \( g \) leads to a full synchronisation and stop of all worker threads. To execute \( g \), these worker threads are started, compute \( g \) and terminate. Back in \( f \), the multithread environment has to be started up again. This kind of artificial et-cell may restrict the performance of the rearranging algorithm as well.

Removing the restriction of st-/mt-mode mixed function to be called in et-mode does not solve the problem. As long as only the master thread can
allocate shared memory all worker threads may enter \( g \) as fast as possible, but they have to wait for the memory to store their results. A way out of this dilemma is either to balance the execution of \( \text{st-cells} \), as shown in Section 7.2, or to distribute the memory management, as shown in Section 7.3. A third possibility is automatic function inlining. Performing this technique requires a cost model to avoid the explosion of the code.
Chapter 8

Conclusion

Parallel computation is a very important issue due to the limits of microprocessors structure size and the costs to build such structures on the one hand and the requirements on performance on the other hand. SAC supports its user to create performant programs for parallel execution without the usual complexity concerning communication and execution.

The existing concurrent execution model has presented a first step of implicit parallelisation. The new multi-thread execution model described in this thesis shows a next step to a better parallel performance of shared memory systems. By abstraction from the single parallelised WITH-loops used in the old model to cells of the same way to execute, it enables the possibility to have large areas of parallel execution without synchronisations in-between. The key to realise these cells is split-phase synchronisation in combination with code rearrangement. This rearrangement is possible, because SAC fulfils the Church-Rosser-Property. It allows a code arrangement optimised for parallel execution.

The described execution model partitions SAC programs into three different areas, determined by assignments which either must or should be executed in a specific execution mode. The execution modes of the remaining code are determined by the codes position depending on the assignments of fixed execution modes. This leads to replication and specialisation of the code unclassified before. Thus, the new model is a combination of the advantages of both, the SPMD and the fork/join model, whereas the disadvantages are avoided. The number of needed synchronisations is minimised via a complete rearranging of the code. The rearranging also enables long distances between data dependent assignments. Hence, the new execution model has the potential to reach a better performance than the existing model.
There are various directions for future work. First of all the SAC compiler must be adapted to the described new multi-threaded execution model. Afterwards, extensive testing and the comparison of runtimes between the old model and the new model will show, if the new model satisfies the expectation of a better performance.

The thesis already identifies the consequences of the execution model to the existing memory management system as well. A new adapted memory management is essential to perpetuate parallel execution beyond function borders. This would improve the performance of SAC programs in parallel not only by removing synchronisations but also by increasing the code flexibility to be rearranged.

Memory management in sequential SAC programs performs memory reuse to reduce runtime memory requirements. Rearranging the code in parallel execution influences the reusing mechanism counterproductively. It prohibits several memory reuses by shifting as many memory allocations as possible up in the code to maximise the possibility of being performed if the worker threads need the memory. Therefore, the memory management adds artificial data dependencies to increase the opportunity for memory reuse. A cost model may be developed to optimise this relationship between minimising synchronisations and minimising memory costs.

Last but not least, another direction to continue the work is the way from shared to distributed memory systems. By mapping the new execution model to a common base like MPI, SAC programs would be enabled to run on distributed memory parallel computers as well.
Bibliography


of the 21st International Multi-Conference on Applied Informatics (A1'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria, pages 478–484. ACTA Press, Anaheim, California, USA, 2003.


