Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND **D**YNAMIC **V**IRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT **C**ONCURRENCY **E**NGINEERING

Statistical Optimisation

D12

Due date of deliverable: January 31st, 2012
Actual submission date: January 31st, 2012

Start date of project: February 1st, 2010

Type: Deliverable
WP number: WP5
Task number: WP5a

Responsible institution: HERTS
Editor & and editor's address: Raimund Kirner
University of Hertfordshire, College Lane
Hatfield, AL10 9AB, United Kingdom

Version 1.0 / Last edited by Raimund Kirner / July 15th, 2011

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	01/06/2011	Raimund Kirner	HERTS	Initial version
0.2	01/07/2011	Nga Nguyen	HERTS	code optimisations
0.3	10/07/2011	Clemens Grelck	UvA	box code optimisations
0.4	13/07/2011	Frank Penczek	HERTS	further on optimisations
1.0	15/07/2011	Raimund Kirner	HERTS	all, feedback from HERTS

Reviewers:
Frank Penczek, Sven-Bodo Scholz, Alex Shafarenko, Clemens Grelck

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP5a	Statistical Optimisation	HERTS*, USTAN, UvA, TWENTE

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

Within the **Advance** project we extent key technologies for the development and execution of concurrent programs to deploy the optimisation potential of runtime optimisations based on statistical program information.

In this document we list code optimisations that are suitable to benefit from the availability of a statistical model of extra-functional properties of programs. We consider code optimisations at the coordination level of S-Net, at the box implementation, and at the combination of both.

Contents

Chapter 1

Introduction

The efficient execution of concurrent application on many-core machines or clusters is a non-trivial problem, making it quite challenging to decide code optimisations and placements at compile time. The novel approach of the **Advance** project is to releave the compilation phase by extending the compilation phase into the runtime phase. This allows to exploit statistical information about the program behaviour collected at runtime to iteratively optimise the program code and the program execution.

In this document we list code optimisations to be considered for the work within **Advance** that are suitable to benefit from the availability of a statistical model of extra-functional properties of programs. We consider code optimisations at the coordination level of S-Net, at the box implementation, and at the combination of both.

Chapter 2

Compiler Optimisations at the Coordination Level of S-Net

2.1 Resource Allocating Adjustment

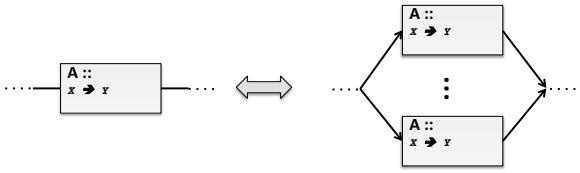

Figure 2.1: Network Transformation in Resource Allocating Adjustment

Description. This transformation creates multiple copies of subnetwork and connects them in parallel as in Figure 2.1. With the current semantic of S-Net, data records are passed to only one random branch of the new subnetwork since all the branches share the same signature. However, with other semantics such as round-robin or priority semantics each branch will take turn to receive the incoming data. Developing more semantics for the split/join operation is a potential task in ADVANCE. In the reversing process, one or more branches can be removed.

Potential Benefits. This technique can be applied to increase the throughput. Reversing process can help to free resources for other purposes.

Applicability Conditions. When a subnetwork is suspected as a bottleneck, this transformation can be considered to allocate more resources for this subnetwork. To preserve the correctness of the application, the original subnetwork should not include any synchro-cell which contains temporal state when the transformation is performed. Extra tasks for split and join operations should be considered. Especially, the join operation can be a new bottleneck.

2.2 Execution Reorder

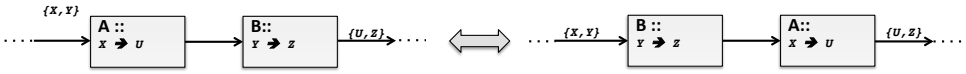

Figure 2.2: Network Transformation in Operation Reorder

Description. The technique reorders the executions of subnetworks which have no data dependence as in Figure 2.2.

Potential Benefits. This transformation can bring some improvements depending on several conditions, for example the execution time of each individual subnetwork, the input rate and the existence of back-pressure.

Applicability Conditions. This technique can be applied only when there is no data dependence between two subnetworks.

2.3 Record Split and Merge

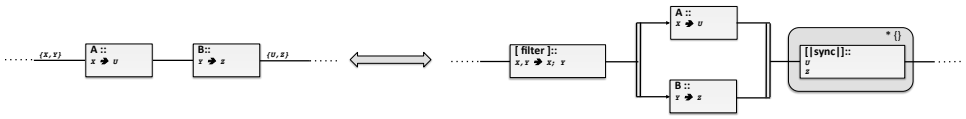

Figure 2.3: Network Transformation in Record Split and Merge

2.3.1 Record Split

Description. The record split technique devides a single input stream into two input streams for two subnetworks which have no data dependence as in Figure 2.3. A filter is used to separate each input record into two records and sent directly to the correspondent box or subnetwork. Two subnetworks are connected by the deterministic split/join operation to avoid wrong matching sets of output records. Two output streams are joined by the end by using a synchro-cell within a serial replication operation.

Potential Benefits. The transformation helps to reduce the latency. However, the overhead of the filter, the deterministic split/join operation, the synchro-cell and the serial replication operation should be noted.

Applicability Conditions. When the requirement in latency is strict, this transformation should be considered. This can only bring benefits if the two subnetworks have long execution time.

2.3.2 Record Merge

Description. This is the reversed process of the above technique. Any pattern as the right-hand side of Figure 2.3 can be applied this transformation.

Potential Benefits. Opposite to the record split transformation, this one helps to reduce the overhead of the filter, the deterministic split/join operation, the syncro-cell and the serial replication operation. Note that despite of serial connection, the transformed subnetwork can still exploit the pipeline concurrency. Obviously, the latency might by increased.

Applicability Conditions. When the requirement of latency is not high, this transformation should be considered to reduce the overhead of non-functional operations (filter, deterministic split/join, synchro-cell and serial replication). The non-deterministic split/join and serial replication operations especially have high overhead.

2.4 Branch Fusion and Fission

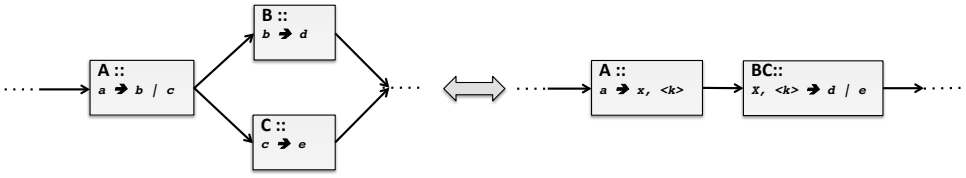

Figure 2.4: Network Transformation in Branch Fusion and Fission

2.4.1 Branch Fusion

Description. This transformation merges two parallel branches whose inputs come from the same preceding subnetwork. As described in Figure 2.4, inputs of two branches B and C comes from A but not at the same time, i.e. an output of A goes to either B or C depending on the data type. In the transformed network, A produces data with type x together with a tag <k>. Type x is an abstract type of b and c while the <k> indicates whether the concrete type of the data is b or c. Two boxes B and C are merged into one box, called BC receiving data x and tag <k>. Depending on the value of <k>, the code of B or C is executed. For example, <k =*1*> indicates that the concrete type of x is b and the code of B is executed. Similarly, <k=*2*> specifies that the concrete type of x is c and the code of C is invoked.

Potential Benefits. When the execution time of boxes B and C is small and the numerous tasks can burden the scheduler, this transformation can be consider to reduce the number of tasks including split/join and branched boxes.

Applicability Conditions. This transformation can be applied for more than two branches with similar technique. The input of all branches should come from one preceding source to avoid the uncertainty of complicated flow inheritance.

2.4.2 Branch Fission

Description. This transformation is the reversed process of the branch fusion. The new subnetwork generated by the branch fusion is transformed back to the original one. It is not possible to split an original box into parallel branches since the box implementation is hidden from the S-Net level.

Potential Benefits. This helps to increase the parallelism which affects the performance. One obvious benefits is to speed the throughput at *A*'s output and therefore help to relieve any back pressure at this point.

Applicability Conditions. When the scheduler is not overloaded by numerous of tasks, this transformation can be considered.

2.5 Explicit Mapping

Figure 2.5: Network Transformation in Explicit Mapping

Description. This technique provides an explicit mapping for a subnetwork instead of using the default mapping from the scheduler as in Figure 2.5. The explicit mapping can be derived by analysing the previous or training runs.

Potential Benefits. The explicit mapping technique can help to improve the performance compared to the default mapping in some circumstances.

Applicability Conditions. This technique is considered when analysing the previous or training runs shows promising improvements. After applied, if it does not improve the performance as predicted, the reversed process can be applied to use the default mapping.

Chapter 3

Compiler Optimisations at the Box Implementation Level

Optimisations at the box implementation level are targeted at improving the efficiency of the computational components within a program. By nature, these optimisations are highly language dependend and every implementation language will require its own set of optimisation techniques and strategies.

3.1 Dynamic Code Optimisation based on Shape Observation and Prediction

In the context of SAC as a box implementation language, the lack of structural information for input data or intermediate results can have a detrimental impact on program runtimes. The efficient execution of applications written in a high-level combinator style requires, for instance, advanced loop fusion techniques [4]. Their effectiveness, however, crucially depends on the level of static knowledge about the data involved. Even advanced symbolic analyses [7, 1], although often successful, cannot fully eliminate this dependency. Ultimately, only radically specializing generic programs to actual input data guarantees sufficient static knowledge for a compiler to produce highly efficient code. Yet, for many applications, the increased size of executables and additional compilation time due to the specialisation for large numbers of potential inputs is prohibitive.

Instead of producing a range of specialized versions at compile time, we delay program specialization until runtime. The key ideas are to run the SAC compiler asynchronously on a separate core, and to employ existing mechanisms in the compiler to trigger and to implement the actual code modifications. SAC advocates shape- and rank-generic programming on multidimensional arrays: SAC supports functions that abstract from the concrete shapes (extent along dimensions) and even from the concrete ranks (number of dimensions) of argument arrays. Furthermore, functions yield result arrays whose shape and rank are determined by some com-

putation in the function itself. Depending on the amount of compile time structural information the type system of SAC distinguishes three classes of arrays that induce three different runtime array representations: From non-generic arrays whose structure is fully known at compile time, via shape-generic arrays where only the rank but not the extent of each dimension is known, up to rank-generic arrays that have a fully dynamic shape (see Fig. 3.1).

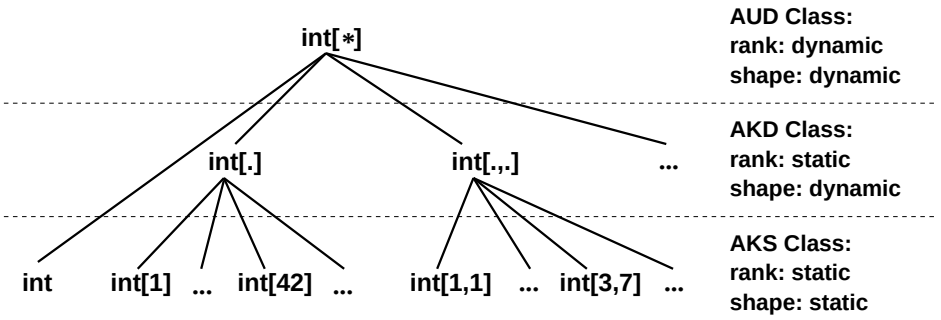

Figure 3.1: Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

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

The benefits that this genericity offers come at a price. In comparison to non-generic code the runtime performance of equivalent operations is substantially lower for shape-generic code and again substantially lower for rank-generic code [8]. The reasons are manifold and their individual impact operation-specific, but three categories can be identified notwithstanding: Firstly, generic runtime representations of arrays need to be maintained, and generic code tends to be less efficient, e.g. no static nesting of loops can be generated to implement a rank-generic multidimensional array operation. Secondly, many of the SAC-compiler's advanced optimisations [3, 5] are just not as effective for generic code because the necessary code properties to trigger certain program transformations cannot be inferred. Thirdly, in automatically parallelised code [2] many organisational decisions must be postponed until runtime and the ineffectiveness of optimisation leads to excessive numbers of synchronisation barriers and superfluous communication.

In order to reconcile the desires for generic code and high runtime performance, the SAC-compiler aggressively specialises rank-generic code into shape-generic code and shape-generic code into non-generic code. However, regardless of the effort put into compiler analyses for rank and shape specialisation, this approach

is fruitless if the necessary rank and shape information is simply not available at compile time.

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

In order to illustrate the shape-generic high-level programming style typical of SAC, we shall consider a small case study: generic convolution with iteration count and convergence check. This computationally non-trivial and application-wise highly relevant numerical kernel fits on half a page of SAC code; Fig. 3.2 contains the complete implementation.

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

The function convolution is defined in lines 7–21; it expects three arguments: an array A of double precision floating point numbers of any shape and any rank (numbers of dimensions), which is the array to be convolved, a double precision floating point number epsilon that defines the desired level on convergence and, last not least, an integer number max_iterations that is supposed to prematurely terminate the convolution after a given number of iterations regardless of the convergence behaviour.

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

Within the **do/while**-loop of function convolution we essentially perform a single convolution step that itself is implemented by the function convolution_step defined in lines 23–32. This function expects an array A of double precision floating point numbers of any shape and any rank and yields a new array of the same shape and rank. In our example, we chose cyclic boundary conditions as exemplified by the use of the rotate function from the SAC standard array library. In fact, the function rotate(index, offset, array) creates an array that has the same rank and shape as the argument array array, but with all elements rotated by offset index positions along

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

Figure 3.2: Case study: generic convolution kernel with convergence check

array axis (or dimension) index. With the **for**-loop in lines 27–29 we rotate the argument array twice in each dimension, by one element towards decreasing and by one element towards increasing indices. Each time we combine the rotated arrays using element-wise addition, as implemented by an overloaded version of the + operator. In essence, this implements a rank-invariant direct-neighbour stencil operation, i.e., in the 1-dimensional case we have a 3-point stencil, in the 2-dimensional case a 5-point stencil, in the 3-dimensional case a 7-point stencil and so on.

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

Coming back to the definition of the function convolution we may want to have a closer look at the loop predicate of the **do/while**-loop in line 18. We continue as long as we neither detect convergence nor the maximum number of iterations is reached. While the latter requires a simple comparison on integer scalar values, the former makes use of the generic convergence test defined in lines 34–37. The function is_convergent checks whether for all elements of the argument arrays new and old the absolute value of the difference is less than the given convergence threshold epsilon. This function definition is a nice example of the SAC programming methodology that advocates the implementation of new array operations by composition of existing ones. All four basic array operations used here, i.e. element-wise subtraction, element-wise absolute value, element-wise comparison with a scalar value and reduction with Boolean conjunction (all), are defined in the SAC standard library.

3.2 Performance Differences for Different Levels of Generality

Fig. 3.3 shows the essential components of the SAC compiler tool chain. Following the inevitable lexical and syntactic analysis, we first do a *functionalisation* of the intermediate code. In this compiler phase most of the imperative-looking features of SAC like for instance C-style branches and loops are converted into properly functional conditional expressions and fully-fledged tail-recursive functions, respectively. The following type inference and specialisation compiler phase infers as concrete as possible array types based on static analysis of the code. The by far largest and most important part of the SAC compiler tool chain, however, are the high-level optimisations. This is the heart of the compiler. We assemble a large number of compiler optimisations and aim at computing the fixed point of intermediate program representation with respect to the various program transformations. Among them are many text book compiler optimisations, such as function inlining or common subexpression elimination, but likewise a large number of SAC-specific

optimisations geared towards the generic array programming context.

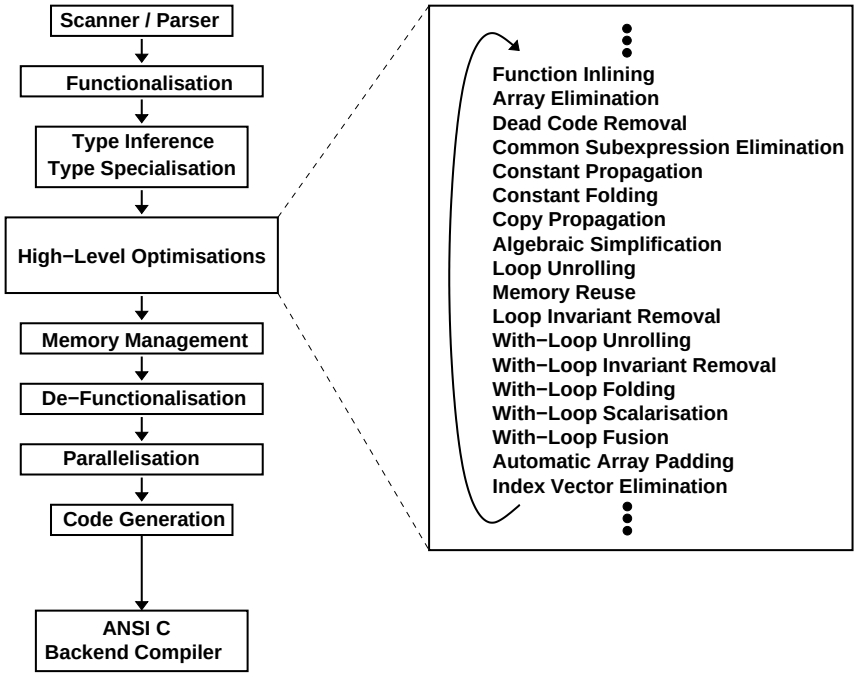

Figure 3.3: Organisation of the SAC compiler tool chain

Following an aggressive, typically large-scale reorganisation of intermediate code by the various compiler optimisations, we have two fundamental lowering steps. The memory management compiler phase introduces symbolic memory allocation and de-allocation as well as reference counting operations into the code [6]. The *defunctionalisation* compiler phase revokes many of the transformations made earlier; for instance, tail-recursive functions are again transformed into loops. The resulting intermediate code may on demand automatically be parallelised [2] before we finally generate ANSI C code. Any ANSI-compliant C compiler can then be used to obtain executable binary code.

We highlight the architecture of the SAC compiler tool chain here to illustrate how little it resembles a light-weight just-in-time compiler. Instead it is a heavy-weight highly optimising compiler and designed for exactly this purpose: generating efficient code, not efficiently generating code. Of course, one could accelerate compilation by reducing the amount of optimisation or even switching off optimisation entirely, but that runs counter our purpose. We particularly aim at exploiting the optimisation capabilities with the proposed adaptive compilation framework, just at application runtime.

Going from bottom to top, Fig. 3.4 illustrates the different runtime representations used by SAC for arrays of (statically) known shape (AKS), arrays of known

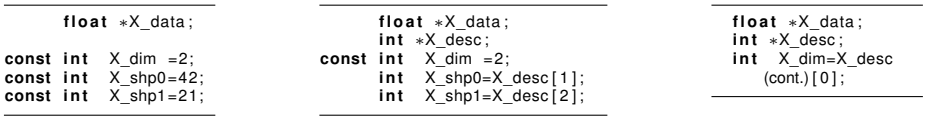

Figure 3.4: Different runtime representations for arrays of (statically) known shape (AKD) left, arrays of known dimension (AKD) center and arrays of unknown dimension (AUD) right

dimension (AKD) and arrays of unknown dimension (AUD) for the example of a 42×21-matrix of single precision floating point numbers X. In each case we can identify the data vector X_data. In the AKS case we can store all structural information as constants, i.e. the rank of the array (X_dim) and the extent along the two dimensions (X_shp0 and X_shp1). Whenever structural information of the matrix X is needed further in the code the C compiler can immediately use these constant values in assembly generation. In the AKD case, the data vector X_data must be accompanied by a *descriptor* X_desc that dynamically carries the structural properties of an array together with the data vector around between function invocations. Still, the rank can be stored as a constant and the shape information can be cached in registers within a function context for more efficient access. Last not least, in the AUD case we have the same descriptor as in the AKD case, but here we also need to extract the rank from the descriptor. We may cache the rank information in a register, but since we do not know the number of dimensions at compile time, we cannot do the same for the shape information. Consequently, any access to this information inflicts a costly memory load operation to retrieve the information directly from the descriptor.

In analogy to the different data representations shown in Fig. 3.4 we can also identify an important difference in the generated code operating on such data. As long as we at least know the rank of arrays statically, (irregular) operations on them can still be represented as efficient nestings of **for**-loops. As soon as we are confronted with rank-generic code, we need to mimic a multi-dimensional loop structure by a single loop and costly retrieval of a conceptually multi-dimensional index location from a single scalar index.

The third major source of overhead of generic code as opposed to non-generic code stems from reduced effectiveness of high-level optimisation. As an example, consider our case study code in Fig. 3.2. If we know the rank of the argument array to function convolution_step at compile time (e.g. through specialisation), we can unroll the **for**-loop in lines 27–29 as typically the rank will be a relatively small integer number. As a consequence, the induction variable i in the application of rotate becomes a constant. In conjunction with the already constant rotation offset, we can highly optimise/adapt the rotation operation to the concrete requirement. Furthermore, we can condense the now statically known number of rotations into a single array comprehension (**with**-loop) from which we can generate executable

code that closely resembles an optimised low-level imperative implementation of the convolution step.

The goal is to develop an adaptive compilation framework that dynamically optimises and specialises box source code based on the shape information on data that is observed at rutime. A key design choice in our framework is the separation of the gathering of profiling information that triggers specialisation from the actual runtime specialisation itself. The rationale of this is to keep the implementation of the former as lean as possible to keep the impact on compiled application code minimal. Instead, most of the new functionality will be encapsulated in a dynamic specialisation controller. The running program and the dynamic specialisation controllers communicate with each other exclusively via shared data structures.

Optimisations are triggered based on concretely observed array shapes. Additionally, further predictive optimisations may be triggered based on statistical knowledge that is accumulated during runtime by other parts of the ADVANCE tool chain.

Chapter 4

Compiler Optimisations at the Combination of Coordination and Box Implementation

4.1 Box Fusion And Fission

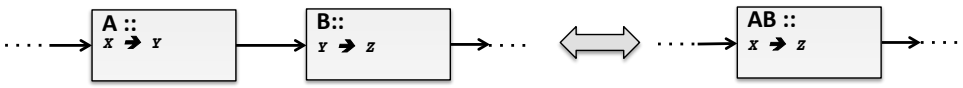

Figure 4.1: Network Transformations in Box Fusion and Fission

4.1.1 Box Fusion

Description. Box fusion is an optimisation technique which merges two boxes into a single one as in Figure 4.1. The code of the new box *AB* is generated by joining *B*'s code after *A*'s. That means the output of *A* is not packed as data records and sent to *B* as originally. Instead, *B*'s code is executed immediately after *A*'s using *A*'s output as the input.

Potential Benefits. This optimisation technique helps to increase the chance to utilise other optimisations from the box compiler. Applying the box fusion optimisation can reduce the communication overhead between two boxes.

Applicability Conditions. Box fusion is always considered when two boxes are mapped on the same core. However, interleaving executions of two boxes might have some benefits, especially when the combined execution time is large compared to the individual ones and/or there are other components are mapped on the same core.

Box fusion can also be applied when two boxes are mapped on different cores

of the same machine or even distributed machines. In this case, the improvement from the box compiler and communication reduction should be superior to the profit from parallel executions.

4.1.2 Box Fission

Description. Box fission is a reversed process of box fusion. The combined box generated by the box fusion process is reverted back to the original subnetwork, i.e. two boxes connected serially. Spitting an original box is difficult since the box implementation currently is hidden from the S-Net level. This can be realistic when appropriate functionalities are supported by the box language.

Potential Benefits. This transformation increases the parallelism which affects the performance.

Applicability Conditions. When there is resource which is not fully utilised, this transformation should be considered. Also, the communication overhead which depends on the resource locations should be counted.

Chapter 5

Conclusion

In this document we investigated how within the **Advance** project the availability of a statistical model of extra-functional properties of programs, such as array shapes or execution cost in conjunction with annotated user expectations and requirements can be used to effectively trigger compiler optimisations. Given the special programming model of advance consisting of a coordination language (S-Net) and component (i.e. box) programming language, we identified compiler optimisations to be applied at different levels: at the coordination language, at the box implementation, and at the combination of both.

Bibliography

[1] Robert Bernecky, Stephan Herhut, and Sven-Bodo Scholz. Symbiotic expressions. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages (IFL'09), South Orange, NJ, USA, Revised Selected Papers*, volume 6401 of *Lecture Notes in Computer Science*, pages 107–124, Berlin, Heidelberg, 2010. Springer-Verlag.

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

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

[4] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SAC. In G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current and Future Issues of High-End Computing, International Conference ParCo 2005, Malaga, Spain*, volume 33 of *NIC Series*, pages 859–866. John von Neumann Institute for Computing, Jülich, Germany, 2006. [ISBN 3-00-017352-8].

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

[6] Clemens Grelck and Kai Trojahner. Implicit Memory Management for SaC. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, pages 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics, 2004. Technical Report 0408.

[7] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In Olaf Chitil, Zoltan Horváth, and Viktória Zsók, editors, *19th International Symposium on Implementation and Application of Functional Languages (IFL'07), Freiburg, Germany, Revised Selected Papers*, volume 5083 of

Lecture Notes in Computer Science, pages 254–273, Berlin, Heidelberg, 2008. Springer-Verlag.

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