Project no. 248828

ADVANCE

Strategic Research Partnership (STREP)
ASYNCHRONOUS AND DYNAMIC VIRTUALISATION THROUGH PERFORMANCE
ANALYSIS TO SUPPORT CONCURRENCY ENGINEERING

Utilisation of Statistical Runtime Feedback based on Prototypical Implementation
D18

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: WP5c

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 / January 30th, 2012

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	08/01/2012	Raimund Kirner	HERTS	Initial version
0.2	15/01/2012	Nga Nguyen	HERTS	Code optimisations with S-Net
0.3	17/01/2012	Frank Penczek	HERTS	S-Net optimisations, illustrations
0.4	25/01/2012	Clemens Grelck	UvA	Box-based optimisation
0.5	26/01/2012	Frank Penczek	HERTS	further box-based optimisation
0.6	29/01/2012	Clemens Grelck	UvA	adaptive compilation infrastructure for SAC
1.0	31/01/2012	Raimund Kirner	HERTS	Incorporation of feedback

Reviewers:
Frank Penczek, Clemens Grelck, Alex Shafarenko

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
WP2a	Requirements Analysis	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 describe program compilation techniques that contextualise empirical statistical data with the statistical model and program annotated user expectations and requirements.

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.

The software development approach in **Advance** consists of decomposing application software into smaller components, we call *boxes*. The coordination of these boxes, i.e., scheduling and data exchange, is described with a coordination program written in S-Net [8, 12, 4].

The boxes themselves may either be written in SAC [7, 6] or in ISO C. Regardless of the box implementation language, a common feature of box implementation is that they are free of persistent state, which means that the same input always produces the same output of a box. This property of freeness of persistent state enables additional ways of code optimisation at the coordination level. It gives the coordination compiler freedom of replicating whole box to improve performance, without having to look into the box's implementation.

In this document we present three different code optimisation techniques. Section 2 and 3 describe code optimisation techniques at the coordination level (S-Net). In Section 4 we describe a code optimisation technique that works at the box level, in this case in SAC. Section 5 concludes this document.

Chapter 2

Explicit Mapping

2.1 Description of Transformation

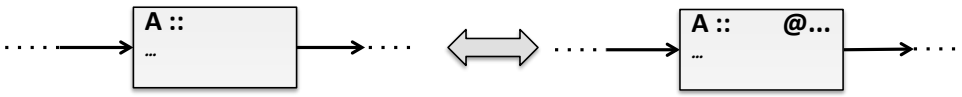


Figure 2.1: Network Transformation in Explicit Mapping

Space scheduling of dynamic and static network structures in S-Net is done by the compiler and the runtime system. When using the LPEL backend for the runtime system, this automatic space scheduling is about mapping tasks to workers in LPEL, where each worker is assumed to run on a different core and tasks are the runtime instances of S-Net boxes.

The automatic space scheduling implemented so far does not consider any extra-functional properties of the different S-Net boxes. Given more knowledge on the extra-functional properties may enable to chose a more efficient space scheduling of the different tasks. The automatic space-scheduling of S-Net tasks can be overwritten by explicit mapping annotations to S-Net boxes or subnetworks.

The Explicit Mapping code optimisation of S-Net programs is thus the addition or changing of mapping annotations in the S-Net source code as illustrated in Figure 2.1. The explicit mapping can be derived by analysing the log of previous or training runs. This transformation can help to improve the performance compared to the default mapping in some circumstances.

2.2 Triggers of Transformation

This technique is considered when analysing the previous or training runs shows promising improvements. The log analysis includes the load balance and/or and runtime component dependence among resources. As each network component is

translated into one task at the runtime, the runtime component dependence can be called as task dependence.

The load balance can be expressed by either the execution time or the waiting time of resources. Depending on the estimated execution time of each task, the allocation should be reconfigured so that each resource is assigned similar amount of work.

The task dependence provides more detailed about the waiting between tasks, for example, waiting for the data from input streams or waiting for the availability of output streams. The task dependence is therefore dynamic and rely on the input rate, the execution rate of each task, the mapping and the time scheduling. In case which these elements are stable or predicted, i.e., the task dependence is similar in training runs and actual runs, it should be considered as a condition to trigger the optimisation.

A new task mapping can be evolved from the load balance and task dependence.

2.3 A Case Study: Option Pricing with Monte Carlo

2.3.1 Problem Description

We study the application of using the Monte Carlo method to calculate the option price [1]. The S-Net implementation of the application is shown in Figure 2.2 The box *png* generates random numbers while *vcall* calculates the underlying assets. All these asset values are accumulated by the box *acc* within a serial replication operation. When *acc* receives the last value, it computes the average value and produce the option price by applying the discount factor.

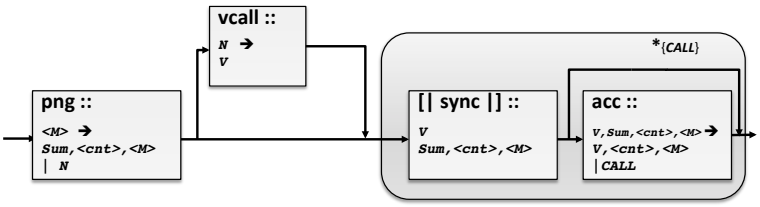


Figure 2.2: The Monte Carlo Option Pricing network

The ultimate goal is to provide the option price with 10^6 price paths. However, the application is first observed with 50 paths to be optimised before being executed with the real number. The experiment is performed on a machine with 2 dual-core AMD sockets and 8GB of memory.

2.3.2 Log Analysis

The log files are analysis first to examine the load among cores. In this case study, the input rate and execution rate of tasks are similar in both training and actual

Worker	0	1	2	3
WE	45	128	105	152
WT [ms]	10.9	22.0	26.4	22.6

Table 2.1: Waiting events and waiting time of each core

runs. Both load balance and task dependence are therefore examined. The load is reflected by the waiting events (WE) and waiting time (WT) of each core as shown in Table 2.1. The number of waiting events is quite large in sense of 50 price paths. Also, this number is relatively high for core 1, 2 and 3 compared to for 0.

In addition, the log shows that the default mapping spread widely synchro- cell, box *acc* and serial replication operation among the cores. This is not efficient because these tasks are dependent on each other and their execution time is very small. With these facts, the default round-robin mapping will not get benefits from parallelism but increase the communication overhead instead.

2.3.3 Derived Explicit Mapping

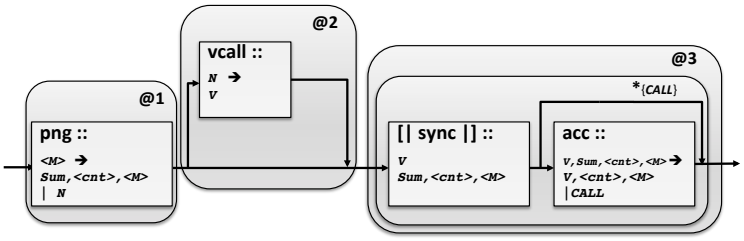


Figure 2.3: The Monte Carlo Option Pricing network

Based on the log analysis on the above section, we propose an explicit mapping in which dependent tasks are mapped into the same core. The detailed mapping is shown in Figure 2.3.

2.3.4 Optimisation Result

In order to evaluate the transformation, two implementations, one with default mapping and one with explicit mapping, are executed with the real input (10^6 price paths). We measure the performance in term of execution time (in seconds) of the whole application. The result is shown in Figure 2.4. The explicit mapping makes an improvement of 188%. This result shows the realistic of the optimisation technique.

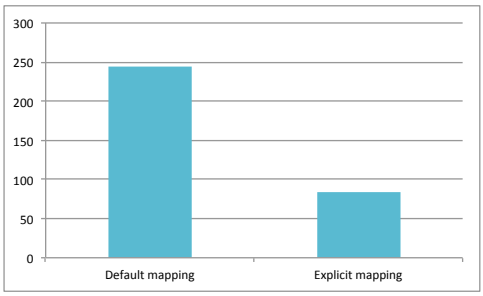


Figure 2.4: The Monte Carlo Option Pricing network

Chapter 3

Resource Allocating Adjustment

3.1 Description of Transformation

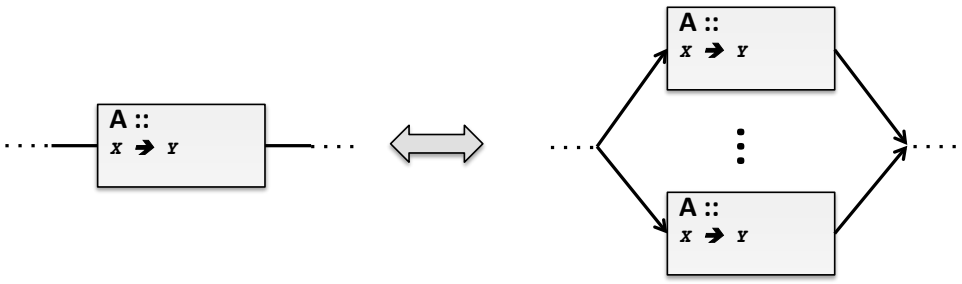


Figure 3.1: Network Transformation in Resource Allocating Adjustment

This transformation creates multiple copies of a box or subnetwork and connects them in parallel as in Figure 3.1. With the current semantics 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. Adding semantics for the split/join operation is a potential task in ADVANCE. In the reversing process, one or more branches can be removed.

3.2 Triggers of Transformation

This technique is recommended when a network component is detected as a bottleneck. Allocating more resources to the bottleneck component can reduce the congestion and therefore improve the throughput. The reversed process can be applied when elements which establish the congestion no longer exist. In this case, additional resources can be free for other purposes.

Congestion is detected by examining the input streams of the suspected component. From the statistical information of stream states, the task dependence of the suspected component and its proceeding components can be derived. Observing the state of communication streams, we can obtain the frequency of proceeders have to wait for the suspected component. The high frequency value is an evidence of the bottleneck or congestion problem. This evidence is a trigger for this optimisation technique.

3.3 A Case Study: DES Encryption

3.3.1 Problem Description

The network shown in Figure 3.2 implements a DES cipher application [2]. From a given key, the box *GenSubKey* generates the required key set for each of the 16 rounds of the ciphering process. The boxes of the network implement the three stages of this process: box *InitialP* applies the initial permutation and splits a block of bits into two blocks of equal size. The box *DesRound* implements the actual ciphering that is applied to the two blocks. The *DesRound* box is consecutively applied 16 times to the bit blocks, once for each key in the key set, by means of a $\star$-combinator. The box *FinalP* joins up the two blocks into one cipher text block and applies the final permutations.

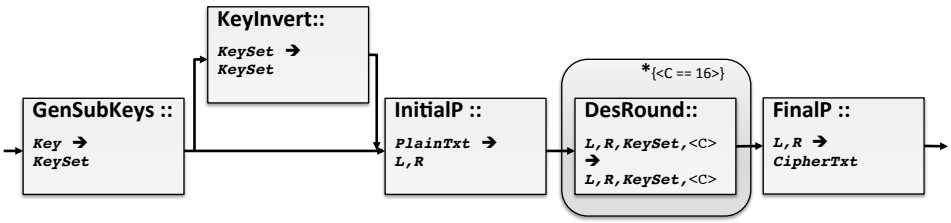


Figure 3.2: Network Transformation in Resource Allocating Adjustment

3.3.2 Stream Observing

With the stable input rate, we observe the input stream of *FinalP* as well as the output stream of the proceeding serial replication operation. Within around 814 milliseconds (1K of data is encrypted), the stream gets full for 11 times and the serial replication operation has to wait for *FinalP* for 144 milliseconds in total. This shows that *FinalP* can be a bottleneck.

3.3.3 Derived Resource Allocation

Based on the above analysis, we propose to create two more instances of *FinalP*, i.e., two more other resources are allocated for FinalP. The transformed network is

shown in Figure 3.3.

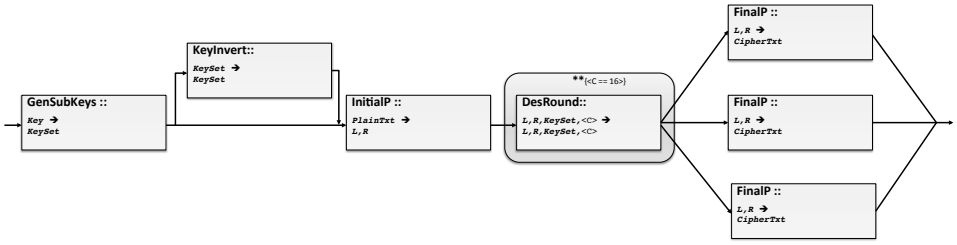


Figure 3.3: Network Transformation in Resource Allocating Adjustment

3.3.4 Optimisation Result

To evaluate the optimisation technique, we perform an experiment to compare the original network (Figure 3.2) and the transformed one (Figure 3.3). The throughput (KB/s) of two networks is shown in Figure 3.4. The throughput has increased from 3.47 KB/s to 3.89 KB/s, gaining 11.85%.

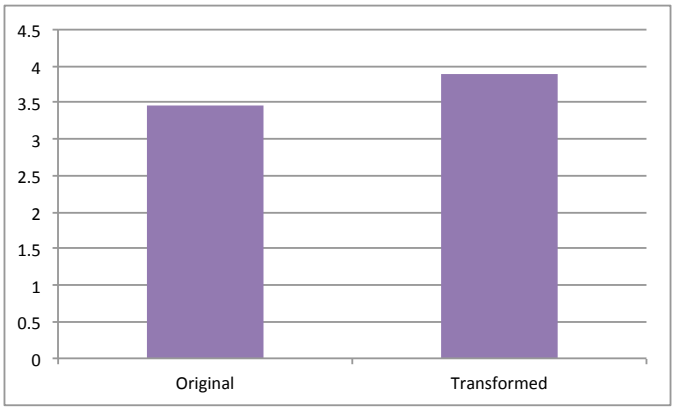


Figure 3.4: Network Transformation in Resource Allocating Adjustment

Chapter 4

Dynamic Code Optimisation based on Shape Observation and Prediction

Programming productivity very much depends on the availability of basic building blocks that can be re-used for a wide range of application scenarios and the ability to define rich abstraction hierarchies. Driven by the aim for increased re-use, such basic building blocks tend to become more and more generic in their specification: structural as well as behavioural properties are turned into parameters that are passed on to lower layers of abstraction where eventually a differentiation is being made. In the context of array programming, as advocated by SAC, the primary box language in the Advance project, such properties are typically array ranks (number of axes/dimensions) and array shapes (number of elements along each axis/dimension). This allows for abstract definitions of operations such as element-wise additions, concatenations, rotations, etc., which jointly enable a very high-level compositional style of programming, similar to, for instance, Matlab.

However, such a generic programming style generally comes at a price in terms of runtime overheads when compared against tailor-made low-level implementations. Additional layers of abstraction as well as the lack of hard-coded structural properties often inhibits optimisations that are obvious otherwise. Although complex static compiler analyses and transformations such as partial evaluations can ameliorate the situation to quite some extent, there are cases, where the required level of information is not available until runtime.

In the course of the Advance project, we have developed a compilation infrastructure that shifts part of the optimisation process into the runtime of applications. Triggered by some runtime observation, the compiler asynchronously applies partial evaluation techniques to frequently used program parts and dynamically replaces initial program fragments by more specialised ones through dynamic relinking. In contrast to many existing approaches, we perform this optimisation in the least possible intrusive way. We use the fully-fledged SAC compiler run on a

separate core. This measure enables us make use of the highest optimisation level, which requires non-negligible compilation time. Furthermore, We use the compiler's type system to identify potential dynamic optimisations. And we use the SAC module system as a facilitator for dynamic code modifications.

For the time being, our dynamic re-compilation infrastructure is used in conjunction with SAC's array type system. The type system (at the moment) is monomorphic in the element type of an array, but polymorphic in the structure of arrays, i.e. rank and shape. Each element type induces a conceptually unbounded number of array types with varying static structural restrictions on array shapes. These array types essentially form a hierarchy with three levels. On the lowest level we find non-generic types that define arrays of fixed shape, e.g. `int[3,7]` or just `int`. On an intermediate level of genericity we find arrays of fixed rank, e.g. `int[.,.]`. And on the top of the hierarchy we find arrays of any rank, e.g. `int[*]`. The hierarchy of array types induces a subtype relationship, and SAC supports function overloading with respect to subtyping.

The subtyping hierarchy exposes the typical trade-off between generality (i.e. code reusability) and performance. While it is desirable to write code on the AKD or preferably on the AUD type level, it is clear that code on the AKS level is much more amenable to compiler optimisation due to additional static knoweledge available to the compiler and very likely to yield better runtime performance. The latter is not only due to more effective compiler optimisations but likewise due to a more complex runtime array representation required by AKD and AUD arrays to dynamically maintain rank and/or shape information.

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 for whatever reason. Data may be read from a file at runtime, or SAC code is called externally from a non-SAC environment via the sac4c foreign language interface, which is always the case as soon as SAC is used as a box language in conjunction with S-Net, the Advance scenario. Here, our adaptive compilation framework takes a crucial role. It incrementally adapts shape- and rank-generic code to the concrete shapes and ranks used in a specific instance of an S-Net network.

The architecture of our adaptive compilation framework is sketched out in Fig. 4.1. A key design choice in our framework is the separation of the gathering of profiling information that triggers specialisation (bottom part of Fig. 4.1) from the actual runtime specialisation itself (the grey boxes in the upper part of Fig. 4.1). Our aim was 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 is encapsulated in the dynamic specialisation controller. The running program and the dynamic specialisation controllers communicate with each other exclusively via two shared data structures: the dispatch function registry and

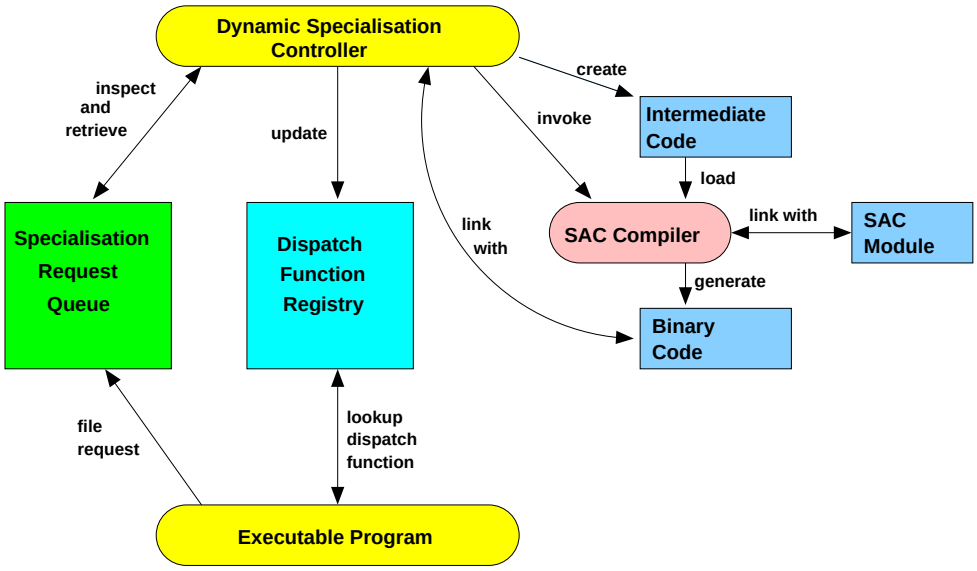


Figure 4.1: Architecture of our adaptive compilation framework

the specialisation request queue (shown as dark boxes in the center of Fig. 4.1). This lean and well-defined interface facilitates the use of multiple specialisation controller instances on the one side and supports multithreaded execution of the program itself on the other side of the interface.

Our design makes use of the existing function call infrastructure within executable (binary) SAC programs generated by our SAC compiler `sac2c`. Such programs (generally) consist of binary versions of shape-specific, shape-generic and rank-generic functions. Any shape-generic or rank-generic function, however, is called indirectly through a *dispatch function* that selects the correct instance of the function to be executed in the presence of function overloading by the programmer and static function specialisation by the compiler. This dispatch function serves as an ideal hook to add further instances (specialisations) of functions created at runtime. Since adding more and more instances also affects function dispatch itself, we need to change the actual dispatch function each time we add further instances. To achieve this we no longer call the dispatch function directly, but do this through a pointer indirection that allows us to exchange the dispatch function dynamically as needed. We call the central switchboard that implements the function call forwarding the *dispatch function registry*.

Our approach is motivated by the observation that the number of different array shapes that effectively appear in generic array code, although theoretically unbounded, often is relatively small in practice. At this point the statistical approach paramount in the Advance project is crucial. Our current implementation of the dynamic re-compilation infrastructure specialises any shape- and/or rank-generic

function, but this only pays off if that function is actually called again with argument arrays that epose the same structural properties. Our next step in the course of the Advance project is to combine our re-compilation framework with a smarter oracle that decides when to dynamically specialise and when not. Continuous monitoring of array properties on the S-Net level plus statistical aggregation will allow us to only perform those specialisations that are beneficial in many cases and to avoid wasting computational resources for specialisations that are unlikely to be needed in the future.

Furthermore, the established dynamic compilation infrastructure opens up an avenue for future research directly connected to the statistical analyses of Advance. For instance, we plan to carry over the approach from specialising for structural properties of argument arrays to adapting code to the specific execution environment it is run on. The availability and type of graphics accelerators and the number and type of CPU cores are again information that normally does not become available until program start. With the existing SAC compilation infrastructure, we can first generate a generic run-anywhere executable code and dynamically adapt it later to the concrete environment it is used on.

So, far this work has led to a peer-reviewed workshop publication/presentation [9] and a journal article [10].

4.1 Implementation

Figure 4.2 gives the pseudo code for function dispatch with dynamic specialisation. Before actually calling the dispatch function retrieved from the registry, we file a specialisation request in the *specialisation request queue*. Apart from information that allows us to uniquely identify the target of the call, i.e., the function name, the module name where the function originates from, etc., this request contains the concrete shapes of all actual arguments to the generic formal parameters of the called function. It is essential here that queuing specialisation requests is implemented as lightweight as possible as this operation is performed for every call to a generic function. To achieve this, we have slimmed the operation down as far as possible. Most information contained in the specialisation request is precomputed and preassembled at compile time. Furthermore, we do not perform any sanity checks during the enqueue operation. These are postponed until the item is later processed by a separate worker thread. This design is geared towards reducing the impact of the proposed adaptive compilation framework on the genuine program execution to a minimum.

Within the same process that runs the *executable program* one or more threads are set apart acting as *dynamic specialisation controllers*. A dynamic specialisation controller is in charge of the main part of the adaptive compilation infrastructure; it always runs asynchronous from the program itself. A dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the specialisation requested already

Listing 4.1: The dispatch function.

```
1  foo_dispatch( arguments) {
2      Collect rank and shape information from runtime descriptors of arguments.
3
4      enqueue_request( "foo",
5                       "moduleBar",
6                       types,
7                       shapes,
8                       pointer into registry);
9
10     foo_ptr = address stored in the registry;
11
12     foo_ptr( arguments);
13 }
```

Figure 4.2: Pseudo code describing the interplay of function dispatch and runtime specialisation.

exists or is currently in the process of being constructed. If so, the request is discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of the specialised function instance to be generated.

The dynamic specialisation controller then invokes the SAC-compiler `sac2c` on the intermediate representation, i.e. the dynamic specialisation controller effectively turns itself into the SAC-compiler. As such, it now starts the standard compilation process for the generated intermediate representation. During this process, the compiler dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SAC module system that was originally developed to support inter-module (compile time) optimisation [11].

Eventually, the SAC-compiler (with the help of a backend C compiler) generates another shared library containing binary versions of the specialised function(s) and one or more new dispatch functions taking the new specialisations into account in their decision. Following the completion of the SAC-compiler, the dynamic specialisation controller regains control.

Before attending to the next specialisation request, two tasks still need to be performed to enable the new specialised code in the running program. Firstly, the controller links the running process with the newly created shared library. As the module name chosen for the stub is unique, this will make a new symbol for the dispatch code of the specialised function available. In a second step, the controller updates the dispatch function registry with the new address of this new symbol for dispatch function(s) from the newly compiled library. As a consequence, any subsequent call to the now specialised function originating from the running program will directly be forwarded to the specialised instance rather than to the generic

version and benefit from (potentially) substantially higher runtime performance without further overhead.

4.2 Experimental Evaluation

The experimental evaluation is based on the execution of a generic convolution kernel on arbitrary arrays as shown in Fig. 4.3. We use an AMD Phenom II X4 965 quad-core system running at 3.4GHz clock frequency. It is equipped with 4GB DDR3 memory, and the operating system is Linux with kernel 2.6.38-rc1.

We ran our generic convolution kernel for a total of six different problem sizes. For each problem size we recorded the execution time of each individual iteration of the convolution using a high resolution real time clock. We uniformly set the number of iterations to 50 while we use a convergence threshold and initial array values which guarantee that we effectively run these 50 iterations. Since we read some initialisation data from a file, the SAC compiler is unable to deduce this information statically, and we effectively evaluate the convergence check in every iteration.

To isolate the effect of adaptive compilation more clearly, we refrain from running the application itself with multiple threads for now and only employ a single instance of the specialisation controller throughout the experiments.

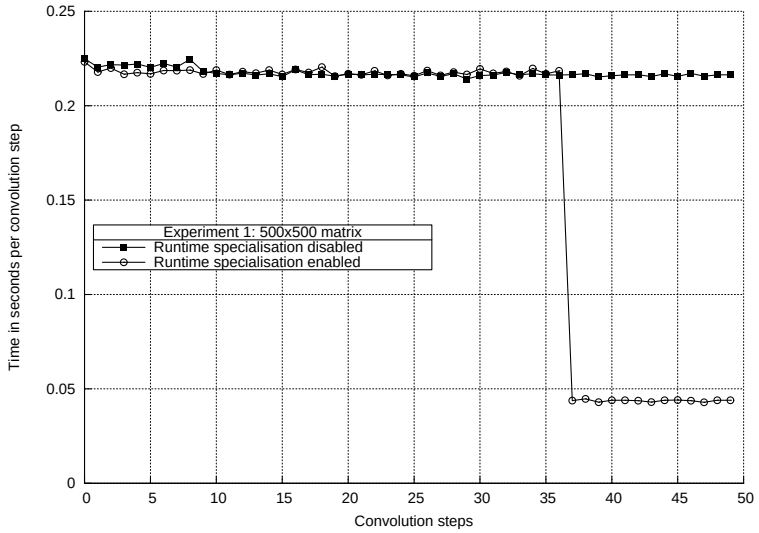
Fig. 4.4 shows experimental results for three different matrix sizes: 500×500, 1000×1000 and $10,000 \times 5,000$. Each experiment is made with and without runtime specialisation enabled. For all three problem sizes we can easily identify a recurring pattern in the runtime behaviour. At the beginning, code with and without runtime specialisation enabled takes about the same time per iteration. In principle, code with runtime specialisation enabled should run slightly slower because it contains additional instructions for spawning the specialisation controller thread, identifying potential specialisation cases and enqueuing specialisation requests. In the given example, this overhead of runtime specialisation seemingly is negligible compared to the computational workload of convolution even for the smallest problem size investigated.

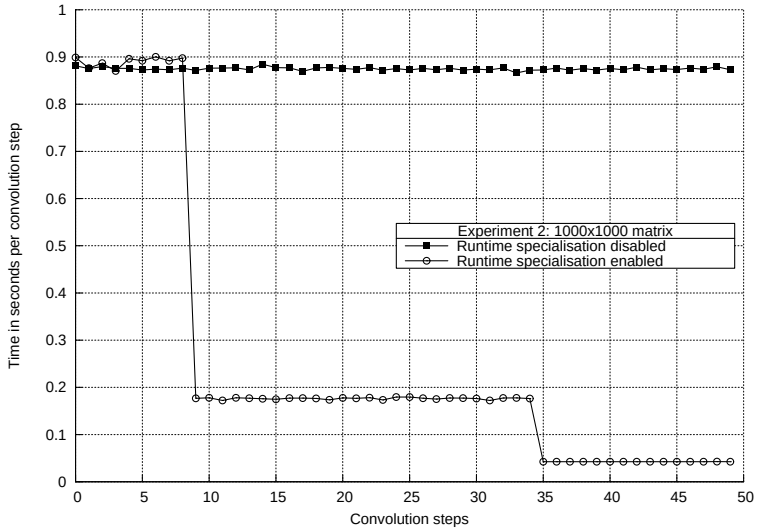
In our case study, the first convolution iteration triggers the first specialisation requests for functions convolution_step and is_convergent. As soon as specialised and, in consequence, far better optimised versions of these functions become available we can identify a dramatic decrease in single iteration runtimes. As the problem size remains constant throughout each individual program run, no further specialisations occur and the shorter iteration runtime remains constant until the end of each program run.

The time it takes to dynamically recompile versions of the functions convolution_step (cont.) and is_convergent specifically adapted to the individual experimental settings is, of course, independent of these settings in general and the problem size used in particular. Hence, the larger the problem size is, the less convolution iterations we need to wait until adapted code becomes available. For a 500×500 matrix,

```
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 4.3: Generic convolution kernel with convergence check





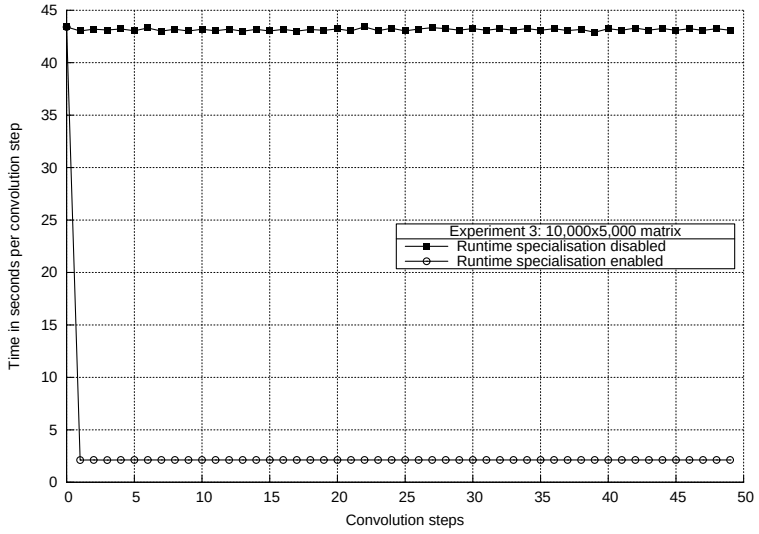
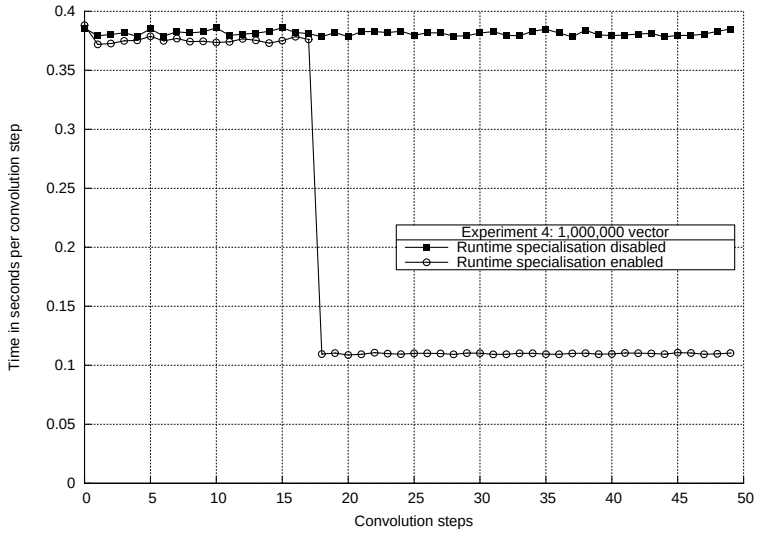


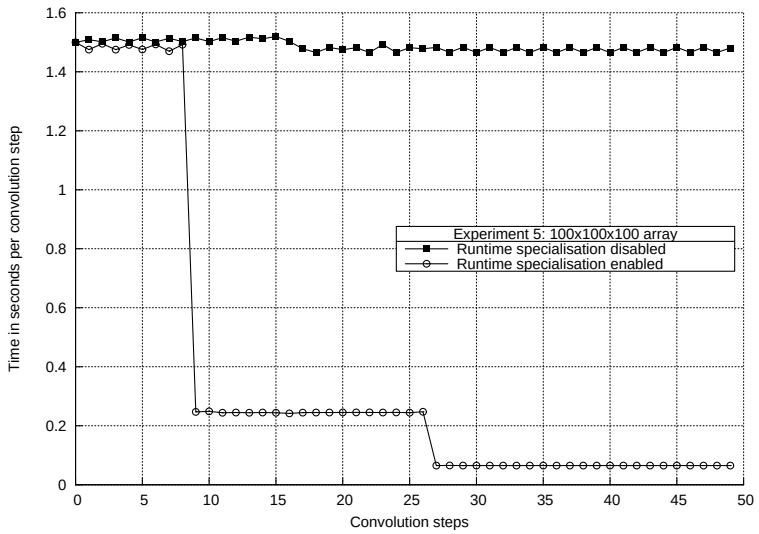
Figure 4.4: Experimental results obtained by applying the case study code of Fig. 4.3 to matrices of different size with runtime specialisation disabled and enabled

it takes more than 35 iterations until the adapted version of the convolution step function becomes available. Since we set the maximum number of iterations to 50 in our experiments, we wait in vain for a specialised version of the convergence check. For a 1000×1000 matrix we only wait for 9 iterations to obtain an optimised convolution step that leads to a more than 5-fold performance increase. After another 26 iterations we also obtain the optimised convergence check, which makes the execution time of a single convolution step drop by another factor of 4. For the largest problem size, a $10,000 \times 5,000$ matrix, a single convolution step is sufficient for the adaptive compilation framework to produce optimised versions of both functions convolution_step and is_convergent. As a consequence, we observe a 20-fold speedup in execution time. These numbers are in accordance with the relative problem sizes as can be expected from a numerical kernel whose computational complexity is linear in the problem size.

One of the remarkable features of our generic convolution kernel is that it can not only be applied to matrices of any size but likewise to arrays of different rank, or number of dimensions. In Fig. 4.5 we show further experimental results obtained from applying the convolution kernel to a vector of $1,000,000$ elements, a tensor of $100 \times 100 \times 100$ elements and a 4-dimensional array of $100 \times 100 \times 100 \times 50$ elements. The experiments confirm our observations in Fig. 4.4. Depending on the size of the workload, we are able to materialize significant improvements within few convolution steps.

How representative is a single benchmark? The proposed adaptive optimisation technique capitalises essentially on two aspects: the performance difference between generic and non-generic code and the unavailability of exact shape information of essential arrays at compile time. In our view, this combination rather is the norm in real-world applications than the exception. The first aspect depends on the actual programming style, of course, but as soon as programs follow the advocated style using composition of building blocks and rich abstraction hierarchies, similar effects as in the convolution example are inevitable. The second aspect is just as relevant. As soon as the building of an application is time-wise and person-wise separated from the running of the application, which again rather is the norm in real-world applications than the exception, the opportunities for static specialisation will be extremely limited. Where this separation does not prevail, e.g. in high performance computing, often applications internally apply the same functions to arguments of different shape in a way that is difficult to deduce statically even if the initial shapes are indeed known. Examples here are the NAS benchmarks MG and FT [5, 3].





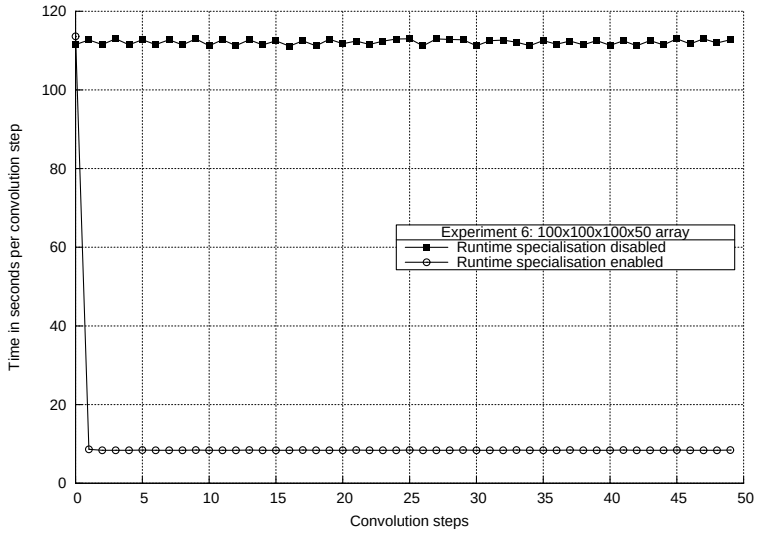


Figure 4.5: Experimental results obtained by applying the case study code of Fig. 4.3 to a vector, a tensor and a 4-dimensional array with runtime specialisation disabled and enabled

Chapter 5

Conclusion

In this document we presented code optimisations of the **Advance** project that allow to contextualise empirical statistical data with the statistical model and the program annotated user expectations and requirements. We have presented several optimisation techniques, some at coordination level, and one directly at box level. We described the statistical data that can be suitable to trigger this optimisations. First experimental data show that the presented optimisations at the coordination language level and at the box level can provide significant performance improvements.

Bibliography

[1] Fischer Black and Myron Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[2] DES. Data Encryption Standard. In *FIPS PUB 46-3, Federal Information Processing Standards Publication*, 1977.

[3] C. Grelck and S.-B. Scholz. Towards an Efficient Functional Implementation of the NAS Benchmark FT. In V. Malyshkin, editor, *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia*, volume 2763 of *Lecture Notes in Computer Science*, pages 230–235. Springer-Verlag, Berlin, Germany, 2003.

[4] C. Grelck and A. Shafarenko. Report on S-Net: A Typed Stream Processing Language, Part I: Foundations, Record Types and Networks. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom, 2006.

[5] Clemens Grelck. Implementing the NAS Benchmark MG in SAC. In Viktor K. Prasanna and George Westrom, editors, *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.

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

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

[8] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[9] Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. An Adaptive Compilation Framework for Generic Data-Parallel Array Programming. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*. Vienna University of Technology, Vienna, Austria, 2010.

[10] Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. *Concurrency and Computation: Practice and Experience*, 2011.

[11] S. Herhut and S.-B. Scholz. Towards Fully Controlled Overloading Across Module Boundaries. In C. Grelck and F. Huch, editors, *16th International Workshop on the Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany*, pages 395–408. University of Kiel, 2004.

[12] A. Shafarenko, S.B. Scholz, and C. Grelck. Streaming networks for coordinating data-parallel programs. In I. Virbitskaite and A. Voronkov, editors, *Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference (PSI'06), Novosibirsk, Russia*, volume 4378 of *Lecture Notes in Computer Science*, pages 441–445. Springer-Verlag, Berlin, Heidelberg, New York, 2007.