Engineering Concurrent Software Guided by Statistical Performance Analysis

Clemens GRELCK [a], Kevin HAMMOND [b], Heinz HERTLEIN [c], Philip HÖLZENSPIES [b], Chris JESSHOPE [a], Raimund KIRNER [d], Bernd SCHEUERMANN [e,1], Alex SHAFARENKO [d], Iraneus TE BOEKHORST [d] and Volkmar WIESER [f]

[a] *Institute for Informatics, University of Amsterdam, The Netherlands*
[b] *School of Computer Science, University of St Andrews, United Kingdom*
[c] *BioID GmbH, Nürnberg, Germany*
[d] *School of Computer Science, University of Hertfordshire, United Kingdom*
[e] *SAP AG, SAP Research Center Karlsruhe, Germany*
[f] *Software Competence Center Hagenberg (SCCH), Austria*

Abstract. This paper introduces the ADVANCE approach to engineering concurrent systems using a new component-based approach. A cost-directed tool-chain maps concurrent programs onto emerging hardware architectures, where costs are expressed in terms of programmer annotations for the throughput, latency and jitter of components. These are then synthesized using advanced statistical analysis techniques to give overall cost information about the concurrent system that can be exploited by the hardware virtualisation layer to drive mapping and scheduling decisions. Initial performance results are presented, showing that the ADVANCE technologies provide a promising approach to dealing with near- and future-term complexities of programming heterogeneous multi-core systems.

Keywords. multicore, software engineering, parallel programming, stream-processing, statistical performance analysis, virtualization

1. Introduction

Engineering concurrent software to run efficiently on a variety of computing systems presents major challenges, in particular outside the domain of scalable regular numerical applications. While today's small-scale homogeneous multi-core processors already challenge conventional software engineering tools and techniques, novel approaches are necessary to make software efficiently and productively utilise massive numbers of heterogeneous cores on a chip that will become available in the near future. Figure 1 provides an overview of the involved challenges with respect to the applications and software development tools targeting multicore hardware. The figure further outlines the work undertaken in the EU-funded ADVANCE project (IST-248828) to address these problems.

[1]Corresponding author: Bernd Scheuermann, e-mail: bernd.scheuermann@sap.com

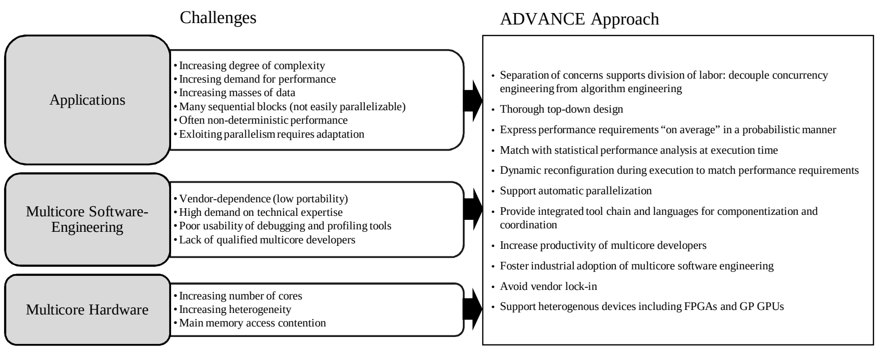

Challenges
ADVANCE Approach
Applications
• Increasing degree of complexity
• Incresing demand for performance
• Increasing masses of data
• Many sequential blocks (not easily parallelizable)
• Often non-deterministic performance
• Exloiting parallelism requires adaptation
Multicore Software-Engineering
• Vendor-dependence (low portability)
• High demand on technical expertise
• Poor usability of debugging and profiling tools
• Lack of qualified multicore developers
Multicore Hardware
• Increasing number of cores
• Increasing heterogeneity
• Main memory access contention
• Separation of concerns supports division of labor: decouple concurrency engineering from algorithm engineering
• Thorough top-down design
• Express performance requirements "on average" in a probabilistic manner
• Match with statistical performance analysis at execution time
• Dynamic reconfiguration during execution to match performance requirements
• Support automatic parallelization
• Provide integrated tool chain and languages for componentization and coordination
• Increase productivity of multicore developers
• Foster industrial adoption of multicore software engineering
• Avoid vendor lock-in
• Support heterogenous devices including FPGAs and GP GPUs

Figure 1 ADVANCE approach addressing the challenges of multicore computing.

The goal is to create an entire tool chain and multicore development framework which is built around three pre-existing, originally independent technologies: SAC, S-NET and SVP. The implicitly data-parallel functional array language SAC [10] combines high productivity software engineering with high runtime performance for rather regular (sub-)problems. The declarative coordination language S-NET [9] turns SAC-implemented application kernels into streaming networks of asynchronous components and, thus, offers a high-level approach to engineering irregularly parallel applications. Last not least, SVP [1] is a hardware virtualisation technology that efficiently maps concurrent tasks onto massively parallel, heterogeneous chip architectures. SVP serves as the common execution layer of SAC/S-NET-implemented application programs.

ADVANCE is more than the integration of existing technologies. A major contribution is a software engineering concept which consistently separates the domains of concurrency engineering and algorithms engineering. Leveraging the experience of domain experts shall enhance developers productivity and foster industrial adoption. On every level, programmers may supply expectations and requirements on extra-functional properties, such as resource utilisation or power consumption, expressed using statistical annotations. These annotations are combined using a static analysis that aims to combine information about average-case throughput, latency and jitter in a statistically valid way.

2. Statistics-Based Concurrent Software Engineering

Deploying massively parallel applications in an efficient way requires specific techniques, which is the focus of the ADVANCE project. To bring the term massively parallel computing to its real meaning, we have chosen to compose a development flow using three key technologies in a combined setting. Algorithmic programming is done with a functional programming language providing implicit data parallelism. This is complemented by concurrency in the large via a coordination language that composes the components of algorithmic programming into a streaming application. The smooth operation on a heterogenous multi-core platform is managed by a virtualisation layer. The approach we introduce here uses statistical methods to drive such a concurrent stream-processing system design in a resource-efficient way.

Figure 2 provides an overview of the ADVANCE Systems Architecture. The top row shows the *compilation route*. Source code in the form of annotated S-Net/SaC pro-

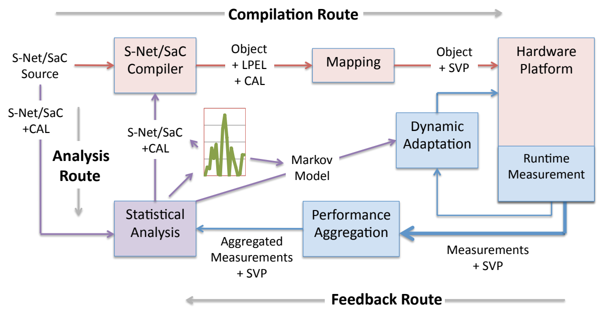


Figure 2 Overview of ADVANCE Systems Architecture

grams is compiled to annotated object code, which is then mapped onto a heterogeneous hardware platform using the SVP virtualisation layer. Runtime profiling information is used to support dynamic adaptation/re-mapping. It is also used to aggregate performance information for specific virtual hardware (described in SVP). This forms the *feedback route* shown on the bottom of the diagram. Finally, these aggregated measurements are combined with user-supplied annotations in S-Net (in the form of CAL, described below). Statistical analysis then generates additional CAL annotations for the S-Net/SaC compilers and a Markov model assisting the dynamic adaptation.

2.1. SAC — Single Assignment C

SAC [10] is a strict, purely functional programming language that combines a C-style syntax with high-level support for processing multi-dimensional stateless arrays. Despite using *highly abstract* notations for array-based computations, the SAC design is geared towards generating *highly efficient* executable code, thus combining high productivity in software engineering with high performance in program execution. Furthermore, the data parallel nature of SAC facilitates fully compiler-directed parallelisation for multi-core/multi-processor systems [6] as well as for NVidia graphics accelerators [12]. Fig. 3 illustrates the high level of abstraction of SAC array programming through a generic convergence check. Essential SAC features such as call-by-value parameter passing for arrays, fully automatic memory management, architecture-agnostic programming and fully automatic parallelisation for different architectures set SAC apart from most other array-oriented languages (e.g. Fortran-90, ZPL [3], CAF [17], UPC [21] or Chapel [4]).

```
bool converge (double[*] old, double[*] new, double eps){
  return( all( abs( old-new) < eps));
}
```

Figure 3 SAC convergence check: The function *converge* accepts argument arrays of any shape including any number of dimensions (type *double[*]*); its definition makes heavy use of pre-defined array operations from the SAC standard library, namely element-wise extensions of standard scalar operators and functions as well as conjunctivec Boolean reduction (*all*).

2.2. S-NET

S-NET [9] is a declarative coordination language that turns both legacy sequential functions and implicitly parallel code into streaming networks of asynchronous stateless components. Each component, or *box* in S-NET terminology, is connected to the rest of the network by two typed streams: a single input stream and a single output stream. Messages are organised as collections of label-value pairs (records). The operational behaviour of a box is characterised by a stream transformer function that maps a single record from the input stream to a possibly empty sequence of records on the output stream. Boxes execute fully asynchronously: as soon as a record is available on the input stream, the box starts computing and, potentially, producing records on the output stream.

Streaming networks are inductively constructed from boxes using a small combinator language. S-NET identifies four construction principles: serial and parallel (static) composition as well as serial and parallel (dynamic) replication. Fig. 4 shows an example S-NET streaming network. As a pure coordination language, S-NET leaves box implementations to a separate *box language*, SAC in the context of ADVANCE. Implementations of S-NET are described in [8,7] while example applications can be found in [11,18]. We chose S-NET for our research in the context of ADVANCE because of its high-level declarative nature and the fact that unlike many other coordination approaches, S-NET achieves a near-complete separation of concerns between component engineering and concurrency engineering.

```
net example {
  box A (...); box B (...);
  box C (...); box D (...);
}
connect A|B .. C!<i>*<done> .. D;
```

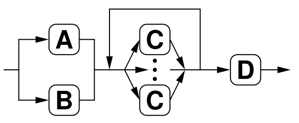


Figure 4 Example of an S-NET streaming network of asynchronous components: The network combinators “dot-dot”, “bar”, “star” and “blink” denote serial composition, parallel composition, serial replication and parallel replication, respectively. Boxes A, B, C and D, are declared locally within network example. For brevity, we omit their type declarations here.

2.3. SVP

SVP is the hardware virtualisation layer used in the ADVANCE project. The motivation for virtualisation in this work is the separation of concerns between the expression of concurrency in the coding of an application and the mapping and scheduling of that concurrency in the execution it. Without this separation of concerns, concurrency issues force the application to coded at a granularity that matches the target chosen, making it not portable to other different targets without recoding it. Our approach is to adapt the mapping of components described in S-NET based on the feedback route identified in Figure 1, so this generality in the capture of concurrency is necessary. The key abstraction in SVP is the concept of *place*, an opaque handle on a resource, which is dynamically acquired and bound to a concurrent section of the code. A place represents a contract with the code to provide a set of execution resources or some execution capability to achieve a specific goal. It may be a thread, a core, a cluster of cores, a function implemented in logic or even a proxy for a given computational requirement. Maximal concurrency is captured in the SVP notation [1] and is transformed by a combination of compilation and run-time system into a granularity that amortises the overheads associated with its implementation on a given target. That run-time system has even been implemented in part in the binary code for the *microgrid* [14].

2.4. *Statistical Performance Annotations*

Where real-time computing requires the worst-case latency to be bounded (e.g. [15]), in our setting we are instead interested in reasoning about the short-term *average-case* behaviour of the system. We must thus design an annotation language that is also capable of expressing the *statistical* behaviour of system components, and devise approaches that can aggregate this information. CAL (*Constraint Aggregation Language*) [20] allows us to write a set of behaviour assertions for each system component, each of which is guarded by some context condition. We use annotations at the component level to reason about the behaviour of the system at the coordination level. For example, The context condition describes the component input, i.e., the properties of input streams and system configurations for which the assertions are valid. For example, we can declare a system component `BOX1` which produces for each input message of type `(a,b)` an output message whose type is `(c,d)`:

```
box BOX1 ((a,b) => (c,d)):
provided
  $a :=: {Type(rank(1), shape($n))}
use
  $n <= 50 => $$T0 :=: $n^2;
  $n >  50 => $$T0 :=: $n^2 / $$nthreads;
     m
end
```

CAL example (see left): The time complexity of the latency (denoted `$$T0`) of the component `BOX1` in case the input object `a` is a vector (`rank(1)`) depending on the length of the vector (`$n`).

In case the length of the vector `a` is at most 50, then the time complexity of the latency is the square of the lengths (`$$T0 :=: $n^2`). However, if the length of the vector `a` is more than 50, then the time complexity of the latency is the square of the length divided by the number of available threads (`$$nthreads`). The ratio behind this CAL annotation can be, for example, that the component processes short vectors serially but switches to a parallel processing for larger arrays, with the intention to avoid the parallelisation overhead for smaller computational tasks. The comment on the annotation overhead, depending on the required level of detail for performance evaluations, the amount of needed CAL annotations might be quite significant. However, at the current stage of research it is not clear how much annotations of useful precision can be produced automatically by the component compiler.

2.5. *Statistical Analysis*

The purpose of the statistical analysis is to determine overall execution costs for systems of components. Knowing the cost of individual components and how they contribute to the overall cost allows us to identify bottlenecks in the system and—ideally—eliminate them by dynamically adapting placement. We are interested in three key cost metrics: i) *latency*, i.e. how long a component takes before it responds to some input; ii) *jitter*, the variation in latencies; and iii) *throughput*, how much can be processed by the system in some given time period.

The ADVANCE approach combines *measurement* with *statistically-based analysis* as follows. We can easily construct probability distribution functions for the latency of a component by repeated measurement of this component, either in isolation, or as part of a larger system. Where possible, we also measure correlations between latencies of communicating components. This enables the identification of those (combinations of) components that contribute most strongly to final output latency and that are thus critical for an optimal placement strategy. Ideally, all the measurement and the statistical processing should be done at run-time so that we can arrive at a good adaptive placement. However, this is unlikely to be practical, since sampling will perturb execution times, and large numbers of measurements may be needed. Instead, we combine off-line measurement of individual components with composition rules to form a statistical model of the cost-behaviour of the system. At run-time, we correlate this model with the actual performance to identify placement optimisations. This approach is aimed at making resource allocation the largest contributor of run-time cost-variations.

The composed model is based on inference rules for (annotated) component types. Let $T = \{\langle\langle a, b\rangle, \delta\rangle\}$ denote the type for a component which maps elements of type a onto elements of type b at cost distribution δ. Variant types can be expressed by adding triplets to a set of types. Every triplet describes a path through the system, where entry and exit points are denoted by element types and the cost for the path is specified. Using this notation, we can define the *transitive type combination* $T_A \otimes T_B$ as $T_A \otimes T_B = \text{flatten}\{\langle\langle a, d\rangle, \delta_A \oplus \delta_B\rangle \mid \langle\langle a, b\rangle, \delta_A\rangle \in T_A \wedge \langle\langle c, d\rangle, \delta_B\rangle \wedge b = c\}$. This can be intuitively understood as the combinations of all paths through the networks A and B respectively. The cost combinators flatten and $\oplus$ must take into account correlations between cost distributions they combine. They represent independent and consecutive path combinations, respectively, and are defined below. With these definitions, we can now describe the inference rules for annotated network types:

(Serial Composition) (Parallel Composition) (Repitition Composition)

$$\frac{\Gamma \vdash A{:}T_A \qquad \Gamma \vdash B{:}T_B}{\Gamma \vdash A \mathbin{..} B{:}T_A \otimes T_B} \qquad \frac{\Gamma \vdash A{:}T_A \qquad \Gamma \vdash B{:}T_B}{\Gamma \vdash A \parallel B{:}\,\text{flatten}(T_A \cup T_B)} \qquad \frac{\Gamma \vdash A{:}T_A \qquad \gamma \subseteq \text{ran}(\text{dom}(T_A)) \qquad \text{free}(n)}{\Gamma, n \vdash A \setminus \gamma{:}\bigotimes_{i=1}^{n} T_A}$$

Since the number of iterations in a repetitive network is not known a priori, a free variable n is introduced to be able to propagate the constraint of the number of iterations outward. There can be predictions for n specified in CAL. If not, at run-time, values of n must be observed to find predictors for future values of n.

Definitions must still be given for the cost combinators used above. The distributions they combine can be correlated both programmatically and because of resource sharing or contention. The latter are two typical examples of reasons why components may show a *correlated deviation* from the cost predicted by the model. Such hidden correlations need to be taken into account. A solution that has recently drawn considerable attention is the use of *copulas* [16]. Copulas are functions that allow joint distributions and their dependencies to be modelled based only on the *marginal probability densities* of the individual components. Given known latencies for two components, we can use a previously-determined copula to predict the overall latency for the combination of the components. We can use a similar approach to deal with throughput. Since jitter is simply the variation in latencies, we can determine this directly from the measurements. The idea of copulas has previously been deployed for worst-case execution time analysis [2]. The novelty in the ADVANCE project is deploying it for average-case time, and using it to predict such costs by *combining* cost information obtained from the underlying components.

The $\oplus$ operator for any two variables is defined as a copula representing the combination of the corresponding property. For latency, the cost of two consecutive networks are added, but for throughput, a meaningful composition of cost metrics is the minimum throughput of the two networks. The copula used for the combination must incorporate the correlations between the combined components. Since flatten is to combine probability distributions for independent paths through the network, it employs the *independence copula*, or Π-copula, viz. $\mathrm{flatten}(T) = \{\langle t, \Pi(\mathrm{ran}(T \upharpoonright \{t\}))\rangle \mid t \in \mathrm{dom}(T)\}$, where $\upharpoonright$ denotes domain restriction.

3. Industrial Applications and Experiments

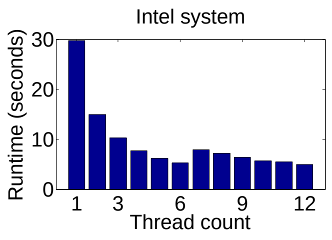
ADVANCE uses four real-world industrial applications to evaluate and demonstrate the proposed concurrent software engineering approach.

Biometric Optimization Framework (BioID): A set of tools and applications to optimize recognition performance and to adapt normalization parameters of a set of pattern recognition algorithms that constitute a multi-modal, biometric authentication system. Large data sets of voice recordings, face and iris images required. ADVANCE allows for calling existing legacy classification modules thereby avoiding time-consuming and costly porting of existing code. Dynamic re-scheduling of multiple concurrent recognition algorithms yields optimal mapping at execution time, improving overall throughput.	*Interventional X-Ray Processing (Philips):* X-ray devices producing a constant stream of images during surgeries. Strict requirements towards accurate hand-eye-coordination, high speed performance, reproducibility, and reliability. Image processing pipelines of boxes (algorithms) with stronlgy data-dependent latency. 'Predictor' boxes process image arrays to work out scalar metrics assessing the latency of 'processing' boxes. ADVANCE uses the output of the predictors and statistical expectations to dynamically schedule low and high-priority tasks to satisfy end-to-end constraints wrt. lateny, throughput and jitter.
Transportation Management (SAP): The NP-hard Vehicle Routing Problem (VRP) seeks to service a number of customers with a certain amount of vehicles. To accelerate optimization by parallel multicore implementation. To use a high-speed version of Ant Colony Optimization [5] following the stream-processing paradigm of ADVANCE. Runtime reconfiguration capabilities of ADVANCE support the implementation of self-adaptive optimization algorithms reacting to dynamic changes in the environment. Concurrent Software Engineering approach is studied in the enterprise software domain fostering industrial multi-core adoption while maintaining developers' productivity.	*Quality Inspection of Textured Surfaces (SCCH):* To identify defects in textures of woven fabrics at scanning speeds up to 300m/min. ADVANCE may reduce the time of developing the whole processing pipeline while saving resources and increasing productivity. The ability to express statistical performance expectations on the average leverages the application where data acquisition and major parts of preprocessing and feature extraction can be computed in a well-predictable way due to the pre-fixed size of filter operations and amount of data transfer known in advance. Additionally, consecutive images obtained during inspection can be processed in parallel, if there exists no dependency.

Next to the the domains mentioned before, ADVANCE leverages such applications which are heterogeneous and complex in nature like e.g. decision supporting and situation aware systems which integrate, correlate, fuse and analysing masses of disparate data resources and streams. Such applications may stem from various industries including e.g. automative, finance, defence and telecommunications and retail. In the following, initial experimental results are reported with the afore-mentioned applications Biometric Optimization Framework and Quality Inspection of Textured Surfaces using the current ADVANCE tool implementations that give a first impression of the scalability of SAC.

3.1. Initial Experiments with the Biometric Optimization Framework

A SAC application to execute an identification test protocol and determine the accuracy of text dependent speaker recognition has been implemented. Mel frequency cepstral coefficients (MFCCs) are used as feature extraction technique, and dynamic time warping (DTW) as the classification method [13]. The calls for computing the classification decision are placed inside a SAC-"with"-loop, rather than a loop with one or more explicitly counting variables. This kind of algorithmic specification enables the SAC compiler to execute the loop by doing concurrent calls on the DTW classifier, which is possible because the relevant identification function is implemented in a thread-safe way. The resulting parallelization is completely automatic and transparent for the programmer, as tasks such as synchronization of spawned threads and dealing with thread pool functionality to control the number of active threads are performed by the SAC-compiler and corresponding libraries. Figure 5 shows the relationship between run times of a single identification rate computation and number of concurrent threads. In the Intel measurement, a minimum of the run time is observed when the thread count is equal to the number of physical cores, 6. With 12 threads, only a small additional improvement can be achieved.



Number of threads	3	6	12	48
Intel system	2.91	5.66	5.97	-
AMD system	2.91	5.55	10.20	25.24

The table shows the speed-up when increasing the number of threads. The Intel system contains an i980x CPU with 6 cores, each supporting "hyper-threading", the AMD system has a total of 48 cores.

Figure 5 Runtimes (left) and speed-up values (right) for identification protocol computation with two multi core systems

As next step for extending the biometric optimization framework, it is straightforward to adapt the classification module itself, such that it makes use of SaC's automatic parallelization as well. Then, it might get impossible to statically make the decisions where to generate concurrency, and what amount of concurrency to generate on different levels, in order to achieve most efficient execution. Instead, the hardware virtualization layer SVP, as described in section 2.3, needs to make these decisions, and it is expected that a feedback route of run time measurements will increase run time efficiency as this makes dynamic adaptations possible (Figure 2). Furthermore, it is desirable to inform the ADVANCE system that the optimization criterion for the biometric application is throughput, which can be achieved through statistical performance annotations as explained in section 2.4. Both run time measurements of the SVP layer and annotations are the input for the statistical analysis, as outlined in section 2.5. By combining these techniques, additional, significant performance improvements of the biometric optimization framework are expected.

3.2. Initial Experiments with Quality Inspection of Textured Surfaces

To demonstrate the auto-parallelization potential of SAC the Perona-Malik anisotropic diffusion filter [19] was benchmarked. The execution time of the filter only depends on the dimension of the input image and not on the content. For our test scenario, we use a

SONY VAIO™PCG-81112M with Intel®Core™i7-740QM, 8GB RAM and a NVIDIA GeForce GT 425M graphic card for benchmarking on GPU. In Table 2, we present the result of applying the anisotropic filter ten times to two images with different input sizes, where we implemented the filter with OpenCV2.2, CUDA and SAC. Implementing an application in SAC allows for executing the application on various hardware environments like FPGA, GPU or CPU. With a similar effort of development, e.g., using C++ and the multithreaded SAC-MT compilation mechanism, currently we are as fast as the OpenCV implementation.

Table 2 Left: Comparison of OpenCV2.2 and SAC implementation of anisotropic filter using SONY VAIO™PCG-81112M. Right: Comparison of manually coded CUDA code and automatic generated SAC-CUDA code using NVIDIA GeForce GT 425M

	$px256 \times 256$	$px4096 \times 4096$
OpenCV	0.03 sec	10.8 sec
SAC-SEQ	0.15 sec	39.3 sec
SAC-MT	0.03 sec	9.7 sec
OpenCV vs. SAC-MT	$1\times$	$1.1\times$

	$px256 \times 256$	$px4096 \times 4096$
SAC-CUDA	0.01 sec	1.8 sec
CUDA-manually	0.04 sec	0.6 sec
Speedup	$0.25\times$	$0.33\times$

The monitoring of the CPU load shows, that the SAC-MT compilation has an optimal processor load on all cores with less memory usage of about 480 MB, where the OpenCV equivalent uses only a single core with a much larger memory usage of about 2.2 GB. In addition, we restricted the execution of the SAC implementation to a single core, with the conclusion that the performance of the SAC-SEQ compilation has poor performance compared to the OpenCV equivalent. Generally, the compiler optimization strategies are designed for multi/many-core systems. Hence, the single core compilation has a overhead, which can be ignored in most cases because a basic requirement for high-performance applications is a multi/many-core environment.

As mentioned in Section 2.1, SAC also provides support for NVidia graphics accelerators, hence we manually implemented and optimized the anisotropic diffusion with CUDA to compare the performance on a graphical processor unit (GPU) with SAC-CUDA automatic generated code. For the moment being, it is not possible to outperform a manual coded implementation by means of SAC-CUDA as we can see in Table 2. The manually coded CUDA application is for the input size of, e.g., 4096×4096 pixels three times faster than the automatic generated GPU compliant SAC-CUDA application. Nevertheless, a re-implementation of the anisotropic filter with high-performance requirements needs definitely higher development costs and programming know-how from experts, where SAC-CUDA allows a flexible time and cost efficient development. However, note that SAC-CUDA is still under development, where we expect further improvements and speedups by means of SAC-CUDA in near future.

4. Conclusion

This paper has introduced the ADVANCE approach to concurrency engineering which uses statistical performance annotations and cost-based information to drive the construction of high-performance heterogeneous multicore software through coordination and hardware virtualisation layers. The core technologies include the CAL aggregation framework, the coordination language S-NET, the box language SAC and the hardware virtualisation layer, SVP. A range of industrial applications are used to evaluate this technology, and we outlined how different application domains may profit from the AD-

VANCE tools and its engineering concept. Our initial scalability results with SAC are promising and give directions for further improvements to reduce compuation and communication overhead. Forthcoming releases and research efforts shall demonstrate that ADVANCE is capable of dealing with near- and future-term complexities of programming heterogeneous multi-core systems and will help reduce development costs to foster multicore adoption in industry while maintaining developer productivity.

References

[1] T. Bernard, C. Grelck, and C. R. Jesshope. On the compilation of a language for general concurrent target architectures. *Parallel Processing Letters*, 20(1):51–69, 2010.

[2] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computing*, 1(2):179–194, 2005.

[3] B. Chamberlain, S.-E. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] W. Cheng, B. Scheuermann, and M. Middendorf. Quick-aco: Accelerating ant decisions and pheromone updates in aco. In *Proc. of the 11th European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP)*, pages 238–249, 2011.

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

[7] C. Grelck, J. Julku, and F. Penczek. Distributed S-Net: High-Level Message Passing without the Hassle. In *ACM SIGPLAN Workshop on Advances in Message Passing (AMP'10)*. ACM, 2010.

[8] C. Grelck and F. Penczek. Implementation Architecture and Multithreaded Runtime System of S-Net. In *Implementation and Application of Functional Languages*, volume 5836 of *LNCS*. Springer, 2011.

[9] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

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

[11] C. Grelck, S.-B. Scholz, and A. Shafarenko. Coordinating Data Parallel SAC Programs with S-Net. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.

[12] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Towards Compiling SaC to CUDA. In *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[13] H. Hertlein, R. Frischholz, and E. Nöth. Pass Phrase Based Speaker Recognition for Authentication. In *Biometrics and Electronic Signatures*, volume 31 of *LNI*, pages 71–80, Darmstadt, Germany, 2003.

[14] C. Jesshope, M. Hicks, M. Lankamp, R. Poss, and L. Zhang. Making multi-cores mainstream - from security to scalability. In *Parallel computing: From multicores and GPU's to petascale*, pages 16–31. IOS Press, 2010.

[15] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond loop bounds: Comparing annotation languages for worst-case execution time analysis. *Software and Systems Modeling*, 2010.

[16] R. B. Nelsen. *An Introduction to Copulas, 2nd Edition*. Springer Series in Statistics. Springer, 2006. ISBN: 978-0-387-28659-4.

[17] R. Numrich and J. Reid. Co-arrays in the next fortran standard. *ACM SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[18] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrière, and E. Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079–2088, 2010. ICCS 2010.

[19] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

[20] A. Shafarenko and R. Kirner. CAL: A Language for Aggregating Functional and Extrafunctional Constraints in Streaming Networks. *ArXiv e-prints*, Jan. 2011.

[21] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.