Towards Self-Adaptive Concurrent Software Guided by On-line Performance Modelling

The ADVANCE Approach

Jaroslav Sykora
Heriot-Watt University, Edinburgh, United Kingdom
J.Sykora@hw.ac.uk

Sven-Bodo Scholz
Heriot-Watt University, Edinburgh, United Kingdom
S.Scholz@hw.ac.uk

Abstract

We present preliminary results on dynamic self-adaptation of streaming networks of concurrent programs (called boxes) to the underlying hardware platform. The monitoring subsystem gathers on-line measurements on the performance of individual boxes and uses the measurements to construct (and improve) performance models of the boxes. Ideally, the models accurately predict the running time of boxes in their next invocations in stream processing. Hence, they can be used by a mapper to optimize the distribution of the limited system resources –processor cores– to the boxes.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Modelling techniques

General Terms Design, performance, load balance.

Keywords S-NET, SAC, LPEL, SVP, auto-parallelization, self-adaptation.

1. Introduction

Contemporary and future compute platforms can deliver massive raw performance, however, the problem is that each platform typically requires an application to be structured and optimized in a different way to take advantage of the hardware potential. Even if we constrain ourselves to just many/multi-core homogeneous general-purpose processor systems the prospect of fine-tuning a large concurrent application to a given hardware configuration (number of nodes, network topology, memory hierarchy) is daunting at the first thought, and impossible at the second when we realize that the effective hardware configuration may change dynamically at runtime due to frequency/voltage scaling of the cores (energy and heat management) and fault-tolerance (cores going temporarily or permanently off-line).

This paper is based on the work carried out during the EU-funded project ADVANCE (IST-248828) [5]. The goals are two-fold. The first is to create a tool-chain that integrates three otherwise independent technologies: SAC, S-NET, and SVP. These technologies are briefly described in Section 2.

The second goal of the project is to research and implement methods for dynamic on-line adaptation of concurrent programs (called 'boxes') to the underlying hardware platform. The idea is to use a light-weight monitoring system to gather performance and other data about the execution of programs in the streaming data-flow S-NET network. At the same time the application developer may supply performance annotations, or 'hints', to express various extra-functional properties of the concurrent programs. The measured data and the annotation would be analysed by a statistical framework and processed by a machine learning system to devise a complex *model* of the system. The model could be in turn used by a mapper to optimize the resource distribution.

2. Infrastructure

2.1 SAC – Single Assignment C

The *Single Assignment C* (SAC) [3] is a strict, purely functional programming language that combines the syntax of C with a high-level support for array programming. The language treats all variables as n-dimensional arrays, and provides overloaded operators and standard library functions to manipulate them. The compiler is geared towards high efficiency of the generated code. 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.

2.2 S-NET

S-NET [4] is a declarative coordination language for describing streaming networks of asynchronous components. Streaming networks are defined using an expression language featuring four network combinators as operators: serial composition, parallel composition, serial replication and parallel replication. With the exception of serial composition, the combinators come in two flavours each: the deterministic versions preserve the order of data on streams, whereas non-deterministic variants trade this property for improved throughput. Two primitive components serve housekeeping and synchronisation purposes. Streams are associated with record types: collections of data where each item is uniquely identified by its name. Structural subtyping on records directs the flow of data through the streaming network.

2.3 SVP – System Virtualization Platform

The *System Virtualization Platform* (SVP) [1] is a hardware virtualization layer used in the ADVANCE project. The purpose is the separation of concerns between the expression of concurrency in the application source code, and the optimal concurrency granularity of the target platform. Without this separation the application developer would be forced to code using the granularity of the platform, making the code not portable to different target platforms.

Copyright is held by the author/owner(s).
FD-COMA 2013 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, part of the 8th International Conference on High-Performance and Embedded Architectures and Compilers, Berlin, Germany, January 21-23, 2013
ACM SIGPLAN conference style.

2.4 CAL – Performance Annotations

The *Constraint Aggregation Language* (CAL) [6] 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.

S-Net components respond to a single input message with zero, one or more output messages, with both the input and the output using a single channel. Since an output message of one component becomes an input message of another, implicative predicates can be chained over connection primitives to aggregate functional constraints over the whole system.

2.5 LPEL – Lightweight Parallel Execution Layer

The *Lightweight Parallel Execution Layer* (LPEL) is a user-space threading library for co-operative execution of *tasks* on *workers*. The fundamental requirement is an efficient scheduling and execution of a huge number of dynamically created tasks, often short-lived, onto a fixed set of processor cores. The processors cores are represented by *workers* that are implemented by standard POSIX threads, pinned to individual processors in the system to improve cache locality and ensure predictable execution.

The LPEL layer provides a dynamic *stream* primitive for task communication. The advantage is that LPEL streams are exposed to the task scheduler and the higher layers, therefore they can be used to reason about task dependencies.

Finally, the LPEL layer provides cheap monitoring services for gathering performance data of individual tasks and streams. The performance measurements are available at run-time for statistical analysis and dynamic adaptation services, and can also be stored in a file for off-line analysis.

2.6 Automatic parallelization in SAC

The SAC compiler can generate multithreaded parallel code executable on common shared-memory Unix-based systems [2]. The compiler exploits massive data parallelism in the implicit array operations of the language.

In SAC the basic building block (not only) for parallel execution is a *with-loop*. A with-loop describes input/output arrays, an index space, and a code block that is executed independently for each point in the index space. The compiler optimization passes will first merge (fuse) with-loops in the source code to enlarge the granularity of the parallel block. Then the index space is partitioned, usually along the outer dimensions, into independent work-tasks[1]. At runtime the work-tasks are put into a pool (if dynamic scheduler is used) and then they are executed concurrently by SAC *bees*. A worker bee is an abstraction layer in the SAC runtime that represents a compute resource, i.e. a processor. In practice, worker bees are implemented by POSIX threads when the program was compiled as a stand-alone application, or by LPEL tasks if the program was compiled for the S-NET environment. There is always a 1:1 mapping of bees to the underlying execution entities, i.e. to the POSIX threads or LPEL tasks.

An important feature is that worker bees are persistent across the whole run-time of the SAC application. The user can choose the number of bees at the program command line, hence allocating the given number of processor cores of the system to the application. Sequential parts of the application are executed by the queen bee (the main program thread) and the other (slave) bees lay dormant. Parallel parts are executed concurrently by all bees of the application. The compiler ensures that there is no nesting of the parallel mode.

[1] Do not confuse with the LPEL tasks, which are a form of threads.

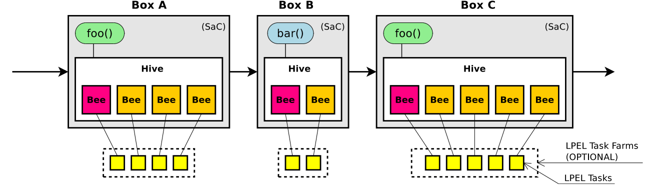

Figure 1. An S-NET pipeline of three SAC boxes with internal bee-hives.

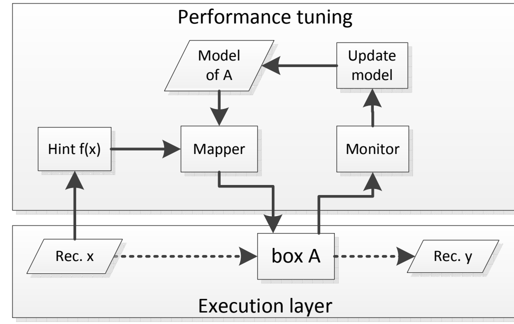

Figure 2. Performance optimization loop.

In the S-NET port of the SAC runtime system the worker bees are persistent across the life time of a single S-NET function box. Each box has its own independent *hive* of bees as shown in Figure 1. The bees are implemented as LPEL tasks and hence they are managed by the common S-NET mapper and scheduler.

3. Box performance modelling

Figure 2 shows our optimization loop. At the bottom there is an execution layer that transports records over S-NET streams and runs the boxes. Above it the performance tuning layer taps into the execution layer to obtain predictive hints $f(x)$ about the expected complexity of the next input record. The hints are combined in a mapper with the dynamic performance model of the box to optimize the execution towards a specific goal, such as latency, throughput, or energy consumption. Global mappers can consider models of many boxes. The execution in a box is monitored and the measurements are used to improve the performance model of the box.

3.1 Definition

Performance model of a box is function τ:

$$\tau_{(\pi)}(p, f(x)) \tag{1}$$

where π are model parameters, p is the number of cores, x is an input record, and $f(x)$ is an optional user-supplied predictor function, $f : X \to \Re$, that evaluates the next input record and predicts its compute complexity. The function τ estimates the box running time (in seconds) for the given number of cores and the input record.

3.2 Amdahl's law

The inverse speed-up S of a program code is limited by Amdahl's law:

$$S(\alpha, p) = \frac{\alpha}{p} + 1 - \alpha \tag{2}$$

where α is the fraction of the program that is parallel, and p is the number of processor cores.

3.3 Basic constant workload model (record-independent)

The *constant workload* model is adequate for boxes that exhibit constant work-load in response to every input record. This basic model ignores record types as the $f(x)$ is missing. The performance model (T_{SEQ}, α) is directly derived from Amdahl's law:

$$\tau_{(T_{SEQ},\alpha)}(p) = T_{SEQ} \cdot S(\alpha, p) = T_{SEQ} \cdot \left(\frac{\alpha}{p} + 1 - \alpha\right) \quad (3)$$

where T_{SEQ} is the sequential time (in seconds) when only a single-core is used, and α is the fraction of the program that is parallel.

3.4 Sequential time estimation

Using CAL annotations the application programmer may specify a predictor function $f(x)$ that estimates the sequential run-time (i.e. the compute intensity) that a given record x will incur in the associated box. Thus, T_{SEQ} is re-defined as a function:

$$T_{SEQ}(x) = f(x) \cdot \beta + \gamma \quad (4)$$

The predictor function $f(x)$ is meant to be specified using the big-O notation, hence the auxiliary parameters β, γ are needed for proper re-scaling. As an example consider a box with sequential run-time proportional to $O(n^3)$. The predictor will be $f(x) = n^3$, where n is extracted from x in an application-specific way, and the estimator is $T_{SEQ}(x) = \beta \cdot n^3 + \gamma$. The lower-order terms (n^2 and n) are ignored.

By substituting Eq. 4 into Eq. 3 we get an improved $\pi = (\alpha, \beta, \gamma)$ performance model that takes into account the record-dependent estimation of the sequential running time:

$$\tau_{(\alpha,\beta,\gamma)}(p, f(x)) = (f(x) \cdot \beta + \gamma) \cdot \left(\frac{\alpha}{p} + 1 - \alpha\right) \quad (5)$$

3.5 Parameter extraction

Model parameters π are extracted from real-time measurements and observations in the running system. The parameters of the basic constant (record-independent) model in Eq. 3 can be estimated from two measurements:

$$\{(p_1, t_1), \quad (6)$$

$$(p_2, t_2)\} \quad (7)$$

$$\text{and:} \quad p_1 \neq p_2 \quad (8)$$

where p_i is the number of cores (the independent variable), and t_i is the observed running time of the box in seconds. From these the α and T_{SEQ} parameters are computed:

$$\alpha = \frac{p_1 p_2 (1 - \frac{t_1}{t_2})}{\frac{t_1}{t_2} p_1 (1 - p_2) - p_2 (1 - p_1)} \quad (9)$$

$$T_{SEQ} = \frac{t_1}{\alpha / p_1 + 1 - \alpha} \quad (10)$$

In the improved record-dependent model in Eq. 5 the extraction of parameters is slightly more elaborate. Given three observations in the form $(p, f(x), t)$:

$$\{(1, f(x_1), t_1), \quad (11)$$

$$(1, f(x_2), t_2), \quad (12)$$

$$(p_3, f(x_3), t_3)\}, \quad (13)$$

$$\text{and:} \quad f(x_1) \neq f(x_2) \wedge p_3 \neq 1 \quad (14)$$

we can directly compute β, γ, α:

$$\beta = \frac{t_2 - t_1}{f(x_2) - f(x_1)} \quad (15)$$

$$\gamma = t_1 - \beta f(x_1) \quad (16)$$

$$\alpha = \frac{p_1 p_2 \left(1 - \frac{t_1 T_{SEQ}(x_2)}{t_2 T_{SEQ}(x_1)}\right)}{\frac{t_1 T_{SEQ}(x_2)}{t_2 T_{SEQ}(x_1)} p_1 (1 - p_2) - p_2 (1 - p_1)} \quad (17)$$

For an unsuitable set of observations (e.g. $f(x_1) = f(x_2)$) we may begin with the simpler constant workload model, and upgrade to the improved model only after acquiring a more diverse set of measurements.

3.6 Discretization and a zone-based estimator

A complementary approach to the model construction is a discretization of the continuous parameter space of the $f(x)$ predictor(s) into a set of non-overlapping ranges or *zones*. An independent model could be constructed in each zone. This way different models could be combined to more accurately predict the performance of a complex box or even a sub-network. However, it is clear that automated construction of such combined models will require more advanced statistical techniques than presented in this paper, and it is left as a future work.

4. Network performance models

We consider here only a serial composition of boxes (a pipeline):

Throughput ϕ of a simple (non-replicated) box is an inverse of its latency:

$$\phi = \frac{1}{\tau} \quad (18)$$

The total latency τ_s [s] of a serial composition of boxes is the sum of individual latencies; the total throughput ϕ_s [1/s] is equal to the smallest throughput in the pipeline:

$$\tau_s = \sum \tau_i \quad (19)$$

$$\phi_s = \text{Min}\{\phi_i\} \quad (20)$$

The pipeline can be optimized for minimal latency or maximal throughput. The global constrain is usually the total number of processor cores. In our analysis we assume the system is running steady-state.

5. Experiments

5.1 Set-up

Figure 3 shows the experimental set-up. The experimental system runs in a loop. Each iteration consists of the following tasks:

1. Getting the prediction $f(x)$ (if available) from the next input record in the queue.
2. Computing the optimal mapping of boxes to processors using the box performance models and the $f(x)$ prediction.
3. Running the boxes.
4. Reading and parsing the box measurements from log files.
5. Updating the box models based on the latest measurements.

In the centre of the figure there is a serial S-Net pipeline of three boxes (A, B, C). The boxes are artificial workloads implemented in the SAC language. Real SAC, S-Net and LPEL runtime systems are used as the infrastructure and for real-time measurements, however, the analysis and mapping subprograms described in this paper were implemented in an external Perl script.

The test-bench boxes A, B, and C in Figure 3 are artificial workload emulators written in the SAC language. Their source code is in

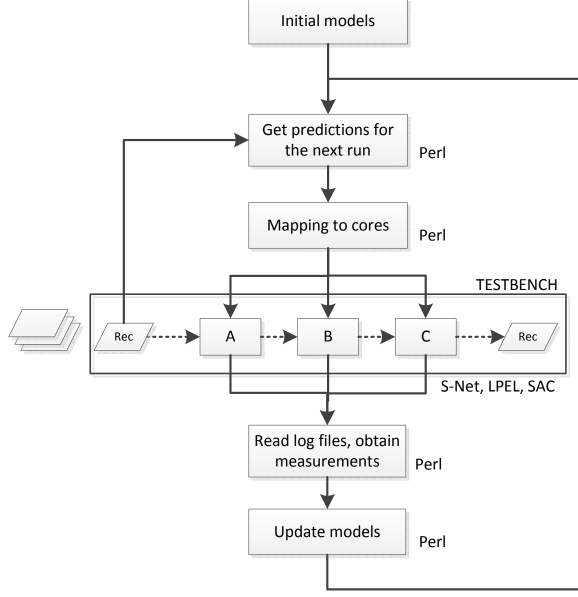

Figure 3. Experimental setup.

Table 1. Box configurations in the experiment

Iterations	Box A $\bar{w}$	Box A $\bar{\alpha}$	Box B $\bar{w}$	Box B $\bar{\alpha}$	Box C $\bar{w}$	Box C $\bar{\alpha}$
1–4	100	0.0	100	0.5	500	1.0
5–9	200	0.0	300	0.5	200	1.0
10–13	300	0.0	200	0.5	600	1.0

Figure 4. The box-code entry point is in the *workload()* function. Its parameter *double[.] V* is an array of three numbers; the parameter is transported as a record via the S-Net streams. The parameter a-priory specifies the work-load the function will emulate at run-time: the total work-load $\bar{w}$ and the fraction of it that is parallel $\bar{\alpha}$.

The total work-load is specified in an ad-hoc *work-unit*. The sequential part $(1 - \bar{\alpha})$ of the work-load is emulated in the *run_seq()* function by a *for*-loop. The parallel fraction (α) is emulated in *run_par()* by the parallel *with*-loop. The SAC compiler will automatically distribute the with-loop to available cores.

The artificial workloads enable us to test the system under different conditions. Of course, the *a-priory* parameters of the emulated workloads are not used in the model construction. As the base work-load is specified in work-units the actual running time depends on the hardware configuration. Similarly, the specified parallel fraction $\bar{\alpha}$ is usually not exactly the same as the observed one in the running system. The discrepancy reflects hidden overheads and/or optimizations in the SAC compiler and run-time. This is however not a problem because what we need is the *models* to accurately predict the *observed* behaviour of the boxes.

The hardware configuration of the experimental system was 8 core Intel Xeon CPU E5506 @ 2.13GHz, 4MB cache size, 12GB main memory, running 64-bit CentOS 6.3.

5.2 Experimental box configurations

Table 1 shows the hidden configuration of work-loads the boxes will emulate in run-time. The $\bar{\alpha}$ parameter is constant for each box over all iterations. We have chosen the box A to be fully sequential ($\bar{\alpha} = 0$), box B half-parallel ($\bar{\alpha} = 0.5$), and box C to be fully-parallel ($\bar{\alpha} = 1.0$). The base work-loads $\bar{w}$ of the boxes are varied over time as shown in the table to simulates different compute complexities of records in real S-Net system.

```
1  /* a unit of work */
2  #define WORK        10485760
3
4  double[WORK] run_seq(double[WORK] a)
5  {
6    s = 0d;
7    for (i = 0; i < WORK; i++) {
8      s = s + a[[i]];
9    }
10   a[0] = s;
11   return a;
12 }
13
14 double[WORK] run_par(double[WORK] a)
15 {
16   s = with {
17     ( [0] <= iv < [WORK] ) : a[iv];
18   } : fold(+, 0d);
19   a[0] = s;
20   return a;
21 }
22
23 void workload( SNet::SNet &hnd, double[.] V)
24 {
25   // V[0] = w = total work-load
26   // V[1] = alpha = parallel fraction
27   // V[2] = garbage
28   alpha = V[1];
29   seq_iters = toi(V[0] * (1.0d - alpha));
30   par_iters = toi(V[0] * alpha);
31
32   a = genarray([WORK], V[2]);
33   for (i = 0; i < seq_iters; ++i) {
34     a = run_seq(a);
35   }
36
37   for (i = 0; i < par_iters; ++i) {
38     a = run_par(a);
39   }
40
41   W = V;  W[2] = a[WORK-1];
42
43   SNet::out( hnd, 1, sacprelude::wrap( W));
44 }
```

Figure 4. Artificial workload emulators in the testbench, written in the SAC language.

5.3 Model construction

We construct the initial performance models of the boxes by handling the first two records (iterations) as a special case. We initially assume that all boxes can be represented by the constant-workload model (Eq. 3). Each box is given one core for the first record, and two cores for the second record. From these two initial measurements (not shown in the tables and plots) the base constant-workload models can be extracted using Eq. 9, 10.

The box work-load configuration values $\bar{w}$ are used as the predictor function $f(x)$. The analysis subprogram starts with the constant-workload models, and switches to the improved (α, β, γ) model when it sees a record with a different $f(x)$.

Figure 5 shows the run-time data over all iterations. The three dashed vertical lines denote the three ‘phases’ from Table 1. The first four plots on the left hand side (a-d) show the evolution of the model parameters over the iteration steps. In the first phase (steps 1–5) the system uses the constant-workload model, hence the β

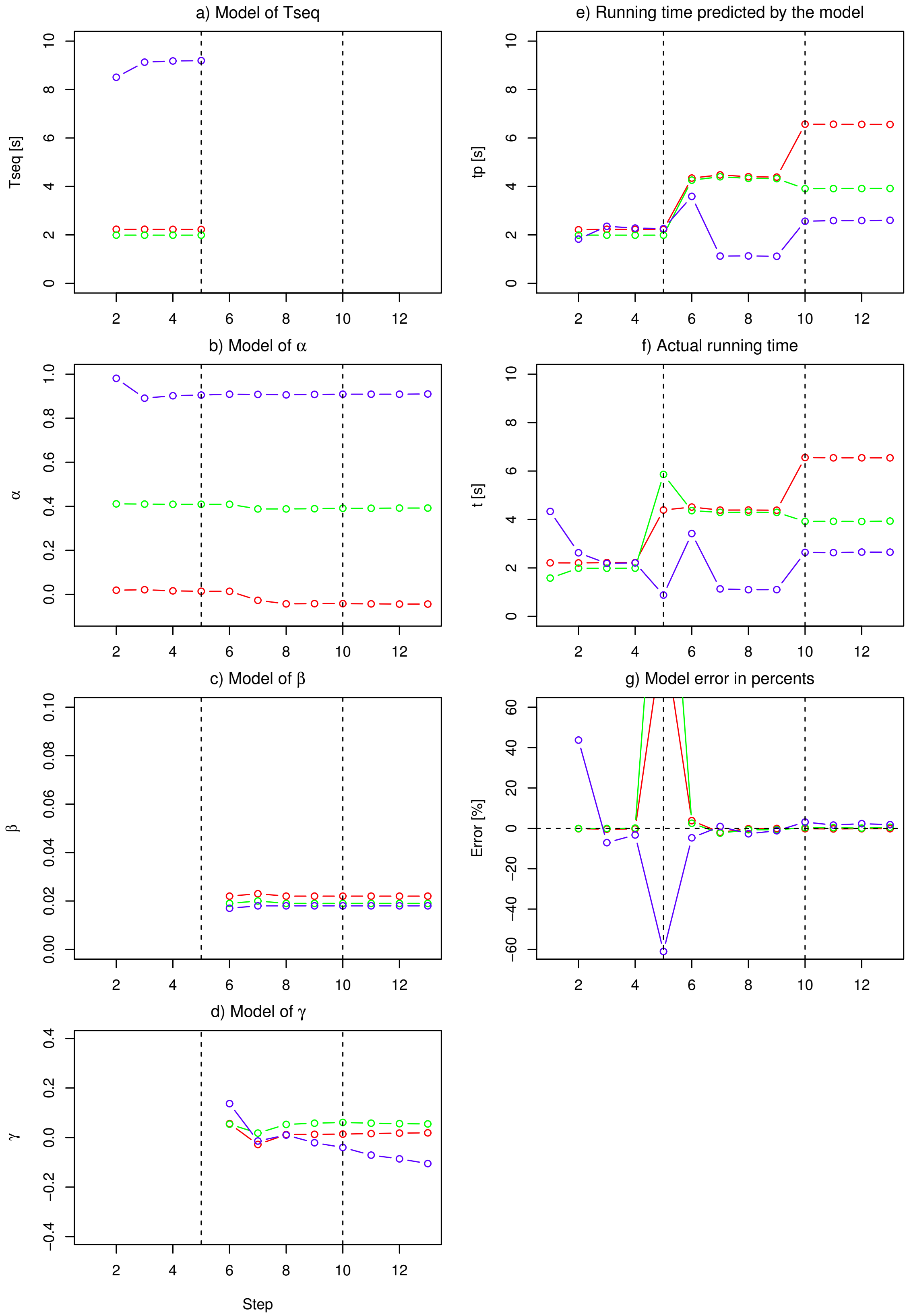

Figure 5. Experimental results. Colours: red = box A; green = box B; blue = box C.

Table 2. Mapper results

Iteration	Opt: Minimal latency			Opt: Maximal throughput			Observed latency
	(p_a, p_b, p_c)	τ_s [s]	ϕ_s [1/s]	(p_a, p_b, p_c)	τ_s [s]	ϕ_s [1/s]	[s]
1	$(1, 2, 5)$	5.642	0.448	$(2, 1, 5)$	6.030	0.452	6.904
2	$(1, 2, 5)$	6.440	0.381	$(1, 1, 6)$	6.577	0.425	6.467
3	$(1, 2, 5)$	6.367	0.391	$(1, 1, 6)$	6.497	0.438	6.487
4	$(1, 2, 5)$	6.343	0.394	$(1, 1, 6)$	6.473	0.443	11.213
5	$(1, 3, 4)$	9.795	0.228	$(4, 3, 1)$	12.196	0.230	12.370
6	$(1, 3, 4)$	10.002	0.223	$(1, 3, 4)$	10.002	0.223	9.885
7	$(1, 3, 4)$	9.868	0.227	$(1, 3, 4)$	9.868	0.227	9.867
8	$(1, 3, 4)$	9.826	0.228	$(1, 3, 4)$	9.826	0.228	9.858
9	$(1, 2, 5)$	12.602	0.152	$(1, 1, 6)$	13.044	0.152	16.028
10	$(1, 2, 5)$	12.630	0.152	$(1, 1, 6)$	13.071	0.152	13.205
11	$(1, 2, 5)$	12.629	0.152	$(1, 1, 6)$	13.071	0.152	13.194
12	$(1, 2, 5)$	12.637	0.152	$(1, 1, 6)$	13.078	0.152	13.206
13	$(1, 2, 5)$	12.640	0.153	$(1, 1, 6)$	13.081	0.153	13.173

and γ parameters (plots c, d) are undefined. In the second and third phases the T_{SEQ} parameter is undefined because the system uses the improved (α, β, γ) model.

Plot (e) on the right hand side of the figure shows the expected running time of the boxes as predicted by the models, and plot (f) shows shows the actual observed time in seconds. Prediction error of the models is in plot (g). The error is the worst in the fifth iteration because at the time the box configuration has been abruptly changed (cf. Table 1). The system adapts in step 6 by switching away from the constant-workload models to the improved models.

5.4 Mapping to cores

Mapping is a separate problem to the model construction. In our setting it means allocating processor cores to individual boxes. There are two basic strategies: to minimize box latency, or to maximize total throughput. As there are only 8 cores and 3 boxes in our experimental system, we simply generate all possible assignments of cores to boxes (p_a, p_b, p_c) with the global constraint $\sum p_i = 8$, and compute the total latencies τ_s and throughputs ϕ_s using the Eq. 19. Table 2 shows the predicted optimal configurations in each iteration, optimized either for the minimal total latency or the maximal throughput. We have used the later in the experimental runs.

5.5 Results and discussion

The prediction error (Figure 5 (g)) is close to zero for all three boxes, except during the fifth iteration when we demonstrate the capacity to adapt the models themselves in response to the availability of the prediction data $f(x)$.

An interesting data point is the iteration No. 10 when we have changed the emulated box workloads again (see Table 1). However, this time the models were able to accurately reflect new running times thanks to the prediction data $f(x)$.

One possible objection to our experimental set-up is that we assume that the parallel fraction α is constant in each box even when the work-load is varied. We make this simplifying assumption because the usual SAC applications (such as image processing) tend to be regular and highly data-parallel. Moreover, it is not clear if the same approach we have used in the T_{SEQ} estimator in Eq. 4, i.e. the use of the big-O notation for the $f(x)$ predictor and the β, γ parameters for its re-scaling, applied to an α estimator, would be transparent and intuitive for the system developer.

6. Conclusion

The presented preliminary results are encouraging. We have used two relatively straight-forward analytical performance models based on Amdahl's law to predict execution times of function boxes in S-NET streaming networks. Free parameters of the models are extracted at run-time from the on-line performance measurements. The models can be augmented by application-level 'hints' in the form of predictor functions that account for the varying compute complexities of individual data records flowing in the network.

Our approach is enabled by three key technologies: (a) the capacity of the SAC language to execute the data-parallel code in a run-time configurable number of cores; (b) the data-flow nature of the S-NET system that has a handle on the repetitive 'firing' of the boxes; (c) the SVP hardware abstraction layer along with the LPEL threading library that provides the monitoring and scheduling services.

Acknowledgments

This work has been supported from EU-funded FP-7 project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" under contract no IST-248828.

References

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

[2] C. Grelck. Shared memory multiprocessor support for functional array processing in sac. *J. Funct. Program.*, 15(3):353–401, May 2005. ISSN 0956-7968. doi: 10.1017/S0956796805005538. URL `http://dx.doi.org/10.1017/S0956796805005538`.

[3] C. Grelck and S. bodo Scholz. Sac: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, page 2006.

[4] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38 (1):38–67, 2010. doi: 10.1007/s10766-009-0121-x.

[5] C. Grelck, K. Hammond, H. Hertlein, P. Hölzenspies, C. Jesshope, R. Kirner, B. Scheuermann, A. Shafarenko, I. te Boekhorst, and V. Wieser. Engineering Concurrent Software Guided by Statistical Performance Analysis. In *15th International Conference on Parallel Computing (ParCo'11), Ghent, Belgium*, 2011.

[6] R. Kirner, F. Penczek, and A. Shafarenko. Compilers must speak properties, not just code - CAL: constraint aggregation language for declarative component-coordination. In *Proc. ACM Workshop on Declarative Aspects and Applications of Multicore Programming*, 2012.