An Infrastructure for Multi-Level Optimisation through Property Annotation and Aggregation

Frank Penczek
University of Hertfordshire, UK
f.penczek@herts.ac.uk

Raimund Kirner
University of Hertfordshire, UK
r.kirner@herts.ac.uk

Raphael Poss
University of Amsterdam, Netherlands
r.c.poss@uva.nl

Clemens Grelck
University of Amsterdam, Netherlands
c.grelck@uva.nl

Alex Shafarenko
University of Hertfordshire, UK
a.shafarenko@herts.ac.uk

ABSTRACT

Optimising software for efficiency on a parallel hardware platform by analysing the performance of the application is often a complex and time-consuming task. In this paper we present a constraint annotation and aggregation system that allows programmers to annotate code by using a dedicated language for describing functional and extra-functional properties, such as for example algorithmic complexity, scaling factors or the number of required cores. The goal is to derive properties of the entire application that are parametrised over characteristics of the execution platform to assist programmers in better understanding the behaviour of an application and to assist the execution platform in making informed mapping and scheduling decisions.

1. INTRODUCTION

Optimising an implementation for efficiency by analysing the performance of an application is often a complex and time-consuming task. Measuring the total execution time of one program execution typically comes for free, for example, by using the `time` command that is available on many systems, but it is the understanding of which factors influence this number that requires thought and effort. Only if we understand how individual parts of a program impact the runtime do we stand a chance to tune the right parameters of the program and its execution platform to improve the overall system performance. When we are dealing with concurrent software the problem becomes even more complex as it is not just algorithmic and computational aspects of the application but factors such as scheduling and mapping may have considerable influence on performance.

Our intention is to provide a means for increasing the efficiency of computation systems, which in our model consists of both programs and the execution platform. We are primarily motivated by the observation that in many software projects the development cycle follows a program-execute-analyse-optimise pattern. This is not necessarily a one way process that takes place at design time only, but can be used equally well as a continuous process that repeatedly assesses the performance of an application at runtime to re-evaluate previous judgements.

This paper is an account of a work-in-progress project for which we present the concept of the overall system design with a focus on the constraint annotation language that acts as the mediator between compilers, runtime systems and observation systems that need to communicate with each other in order to ultimately form a continuous optimisation loop that spans across the entire runtime of an application and potentially across multiple runs as well.

We start with an overall system that follows a coordination approach in which software components are implemented in a "traditional" programming language and a secondary, dedicated coordination language (S-Net) is used to arrange the individual components into an application [1]. The coordination language S-Net treats the computational software components as encapsulated boxes without access to their internal behaviour. All that is exposed to the coordination language is the input requirement of a box, i.e. what input the box requires to carry out its computation, and a specification of the results that the box is producing once the computation invoked on some input has finished. Taking these input/output characteristics into account, the coordination program defines a data-flow graph, which in turn defines the dependencies for each box. The application is executed as a collection of asynchronous components, i.e. the boxes, that start computing as soon as their input requirements are met. The communication between boxes is taken care of by the coordination layer that connects boxes to each other using FIFO channels. Figure 1 illustrates the system design in the absence of our proposed constraint aggregation facilities. In addition to box language compilers and the compiler for the S-Net program, the runtime system is responsible for providing the communication infrastructure at runtime and for mapping the box tasks onto resources of the execution platform. The execution platform is not physical hardware but an intermediate layer, the *virtual hardware platform*. The virtual hardware platform implements task scheduling and placement algorithms and is responsible for executing tasks on physical computing resources.

We propose to extend this system by a central property and constraint aggregation infrastructure. The extended de-

NFSP-DSML '12, October 01 2012, Innsbruck, Austria

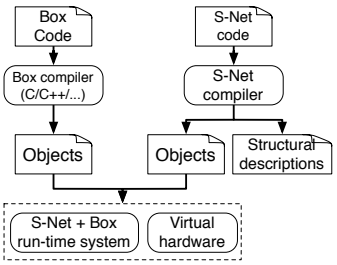

Figure 1: System infrastructure for coordinated program execution

sign is shown in Figure 2. Using a generic constraint and property annotation language all components of the system may expose domain-specific properties that are described in a separate *passport*. The properties described in the passport are centrally collected by an aggregator that uses the information to infer properties of the entire system *at multiple levels*. This process may infer properties statically, i.e. at compile time, as well as dynamically at runtime. Static inference may be used to implement deeper checks for component interoperability within a coordination program. It may be used to initially provide computing resources and memory requirements. It may also enable cross-component optimisation for cases in which the aggregator has been able to infer properties of data objects that are supplied to boxes. This occurs, for example, when sizes of data objects such as vectors and arrays can be statically inferred. At runtime, the system forms a continuous loop with the CAL aggregator being the central information hub. The runtime system and the execution platform monitor the runtime behaviour of computational boxes to check and refine previous annotations regarding time complexity and memory requirements. Similarly, it is also possible to observe certain data object properties that were not available before runtime. This information is relayed by the aggregator to compilers that can take these newly available properties into account for a new round of optimisations when recompiling code and producing new or updated CAL passports. After recompilation, updated versions of boxes may be deployed at runtime, after which the described process repeats itself.

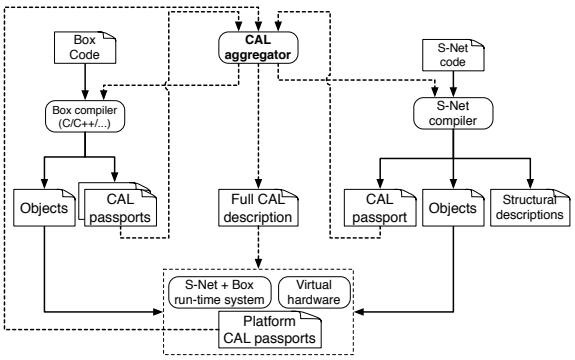

Figure 2: System infrastructure with the CAL aggregator as central information hub

The remainder of this paper is structured as follows. Section 2 introduces the basic concepts of the annotation language that underlies all information propagation in the system. To make the concepts more concrete, section 3 presents a complete example that illustrates how we suggest our proposed technology to be used. Section 4 very briefly surveys related work and section 5 draws conclusions.

2. CONSTRAINT AGGREGATION LANGUAGE

The *Constraint Aggregation Language* (CAL) [4, 7] provides an interface between different stages of program analysis for the description of functional and extra-functional program behaviour. CAL by itself is a rather generic description language; it introduces programming-language specific symbols and annotation on various levels of the software stack. We focus in this paper on the use of CAL to describe the behaviour at the granularity level of functions or methods. In order to be able to infer properties of the overall system behaviour, we need to set the individual passports into context. This is done by establishing data-flow dependencies between functions using the required input and provided output of a function as described by the structural properties of the coordination program. The data-flow graph provides the infrastructure for property propagation between and across the functions that are contained within a system. We assume that any state maintained by a function between multiple invocations is fully identified and visible to the annotation system. This allows us to describe, this carried state as an extra input and output argument to the function and capture it in CAL passports.

To make the CAL concept more tangible, Figure 3 shows the syntax definition for CAL passports. As the syntax definition shows, CAL allows us to write a set of assertions for each function, each of which is guarded by some context condition. For example, the context condition describes the functions input, i.e., the properties of input data objects and system configurations for which the assertions are valid. A property is always represented by a tuple of the form

$$(\text{PropName}, \text{val}_0, \ldots, \text{val}_{n-1}).$$

The symbolic name of the property is the first element of the tuple. All following elements are values to be associated with the property. Declarations in a CAL passport serve two purposes. On the left side we may "query" the properties associated with a data object and bind its values to variables for later use. On the right side, we may attach properties to data objects, either using static values or by referencing variables that we have bound earlier.

For example, consider the C++ function in Listing 1 which computes a value histogram of an input vector based on some quantization factor `qf`. As this function allocates a new vector for its output; it is thus desirable to inform the environment about the shape of the result. We can do this using Listing 2. Here, the predicate states that if the environment can inspect the value of the input "qf," then the predicate holds and the variable `$nh` become bound to the value. When the predicate holds, the shape of the output is known (it is a 1D vector) and its size is given by `$nh`. Note that the declarations between `use` and `end` are conjunctive.

CAL is in its early development stage, brought forward by recent research. More experience is needed in order to make statements about the annotation overhead, which actually depends on the required level of detail, e.g., performance evaluations. Further, it is also an open question of how much information can be produced automatically by other compilers.

Passport ⇒ ⌈ *TempSpec* ⌋ *Interface* ⌈ ≡ ⌈ *Decl* ⌋* ⌋ ;

TempSpec ⇒ ∀ *Id* ⌈ , *TempSpec* ⌋*

Interface ⇒ *Class Id* : *Signature*

Class ⇒ **box** | **resource** | **actor** | **synthesis**

Signature ⇒ *SetBind* ↦ *SetBind*

SetBind ⇒ *Id* . (⌈ *Id* ⌈ , *Id* ⌋* ⌋) ⌈ , *SetBind* ⌋*

Decl ⇒ ⌈ *Guards* ⟹ ⌋ *Statements*

Guards ⇒ *Guard* ⌈ *PredOp Guard* ⌋*

Statements ⇒ *Statement* ⌈ △ *Statement* ⌋*

Guard ⇒ *Relation* | *UseTerm* ∈ *SetExpr*

Statement ⇒ *Relation* | *UseTerm* ∈ *Var* | *SetExpr* ⊆ *Var*
| **provided** ⌈ *Clauses* ⌋ **use** ⌈ *Decl* ⌋* **end**

Clauses ⇒ *Clause* ⌈ , *Clause* ⌋*

Clause ⇒ **let** *Var* := *UseTerm* | **let** *Var* := *SetExpr*
| *BindTerm* ∈ *SetExpr*

BindTerm ⇒ *Var* | *BindTuple* | *

UseTerm ⇒ *ArithExpr* | *UseTuple*

BindTuple ⇒ (*Id* ⌈ , *BindTerm* ⌋*)

UseTuple ⇒ (*Id* ⌈ , *UseTerm* ⌋*)

PredOp ⇒ △ | ▽

Relation ⇒ *ArithExpr RelOp ArithExpr*

RelOp ⇒ ≡ | ≥ | ≤ | ≧ | ≦ | ≠

ArithExpr ⇒ *Atomic* ⌈ *ArithOp ArithExpr* ⌋*

Atomic ⇒ *Num* | *Var*

ArithOp ⇒ + | − | × | /

SetExpr ⇒ *SetDef* ⌈ *SetOp SetExpr* ⌋*

SetDef ⇒ *Var* | { *BindTuple* ∈ *Var* ⌈ ‖ *Guards* ⌋ }

SetOp ⇒ ∪ | ∩

Var ⇒ $ *Id* | $$ *Id*

Id ⇒ ⌈ a-zA-Z ⌋+ ⌈ a-zA-Z | 0-9 | _ ⌋*

Num ⇒ ⌈ - ⌋⌈ 0-9 ⌋+ | *Num* / *Num*

Figure 3: The CAL grammar

Aggregating and finding solutions to a set of constraints is currently approached via unification but we are considering the integration of constraint solving as well. The advantage of using a constraint solver over other possible methods is that in case of insufficiently precise constraints, the constraint solver still guarantees a solution. The solution might be too imprecise, i.e., requiring further concretisation to be useful for compilers and platform configurers, but it can never be incorrect. Consequently, it may well be possible to involve the programmer in the process of iterative refinement, whereby insufficiently tight constraints are identified interactively and further properties solicited in order to narrow them down.

```
vector<float> hist(vector<float>& v, int qf) {
  vector<int> nv(qf);
  float f = qf / max_element(v.begin(), v.end());
  for(int i = 0; i < v.size(); ++i)
     nv[v[i] * f] += 1;
  return nv;
}
```

Listing 1: Example histogram function.

```
box hblur : input.(v,qf) ↦ output.(h) =
provided
    (Value, $nh) ∈ $qf
use
    (Dim, 1) ∈ $h  ∧  (Size, 0, $nh) ∈ $h
end;
```

Listing 2: Simple CAL passport for the histogram function.

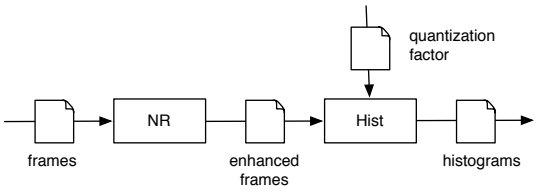

Figure 4: Example image processing pipeline

3. EXAMPLE FROM TOP TO BOTTOM

To illustrate the concepts presented so far, we extend the example from Listings 1 and 2 to an image processing pipeline that produces value histograms from a stream of input frames (Figure 4). The pipeline contains two stages: `NR` for "noise reduction," which applies a blur convolution, and `Hist` which computes a value histogram.

3.1 Programmer-specified passports

The component programmer may specify *box passports* which provide statements about either the *input-output relationship* or the *expected behavior*, or both. For the image processing pipeline above, a first passport for NR may look like Listing 3. In this passport, the input and output are labeled "fi" and "fo," respectively. Within the passport, the statement is predicated on the existence of two terms of the form `(Size, 0, _)` and `(Size, 1, _)` in the input. Assuming the predicate holds, the predicate also binds the variables `$n` and `$m` to the corresponding values in the bound terms. When the predicate holds, the statement applies and defines that the tuples `(Size, 0, $n)` and `(Size, 0, $m)` are included in the output.

We made the tuples explicit in this first example to clarify the matching process; however, it is actually simpler to express shape-generic passports using set operators. For example, Listing 4 expresses that any tuples with labels `Size` and `Dim` in the input are also found in the output.

The intent of passports, beyond defining functional and representational relationships between output and input, is to also make statements about the expected *behavior* of components. A common term to express is the *time complexity* of the component, as a factor of the input characteristics.

An example is given in Listing 5. In this passport, the sig-

```
box NR : input.(fi) ↦ output.(fo) =
provided
  (Size, 0, $n) ∈ $fi, (Size, 1, $m) ∈ $fi
use
  (Size, 0, $n) ∈ $fo  ∧  (Size, 1, $m) ∈ $fo
end;
```

Listing 3: Shape passport for box NR, 2D.

```
box NR : input.(fi) ↦ output.(fo) =
  { (Size, *, *) ∈ $fi } ∪ { (Dim, *) ∈ $fi }
  ⊂ $fo ;
```

Listing 4: Shape passport for box NR, generic.

```
box NR : input.(fi)
  ↦ output.(fo), expectedbehavior.(b) =
provided
  let $shape := { (Size, *, *) ∈ $fi }
              ∪ { (Dim, *) ∈ $fi },
  (CALReduce, ×, (Size), 1, $size) ∈ $shape
use
  $shape ⊂ $fo  ∧  (TimeComplexity, $size) ∈ $b
end;
```

Listing 5: Passport for box NR, with behavior.

nature is extended with a new output set "b" for the properties of expected behavior. The first clause of the statement then defines the alias `$shape` for the set of all `Dim` and `Size` terms in "fi." The second clause uses a special CAL predicate: it binds the variable `$size` to the arithmetic product of the value of the 2nd tuple position of all `Size` tuples in `$shape`, i.e. to the product of array dimensions in the input. Note that tuple positions after the label are numbered from 0. Assuming the predicate holds and binds `$size` successfully, the statement specifies that the output has the same shape and that the time complexity is linear in `$size`, i.e. the input size.

The passport for Hist is specified in the same fashion in Listing 6, which extends Listing 2.

It is also possible to parameterize the expected behavior by run-time parameters. In particular, when the component may exploit available parallelism, the passport can unify with the number of processing units in the execution properties. An example is given in Listing 7. Here, the signature is extended with a new input set "e" for the execution environment properties. The passport then states that either the complexity scales with the available parallelism for input sizes larger than 50, or is linear in the input size if the execution is sequential or for smaller inputs.

3.2 Virtual hardware and operational passports

Virtual hardware is a combination of *actor passports* with special operational semantics, together with a database of *resource passports* which describe hardware properties.

All actor passports have the same signature, defined in Listing 8. In addition to this generic form, actor passports are the only passports in CAL with operational semantics.

```
box Hist : input.(fi, qf)
  ↦ output.(h), expectedbehavior.(b) =
provided
  let $shape := { (Size, *, *) ∈ $fi },
  (CALReduce, ×, (Size), 1, $size) ∈ $shape,
  (Value, $nh) ∈ $qf
use
  (Dim, 1) ∈ $h
  (Size, 0, $nh) ∈ $h
  (TimeComplexity, $size) ∈ $b
end;
```

Listing 6: Passport for box Hist.

```
box NR : input.(fi), execenv.(e)
  ↦ output.(fo), expectedbehavior.(b) =
provided
 let $shape := { (Size, *, *) ∈ $fi }
             ∪ { (Dim, *) ∈ $fi },
 (CALReduce, ×, (Size), 1, $s) ∈ $shape
use
 $shape ⊂ $fo
 (Sequential) ∈ $e  ⟹  (TimeComplexity, $s) ∈ $b
 provided
  (DataParallelism, $p) ∈ $e
 use
  $s ≤ 50  ⟹  (TimeComplexity, $s) ∈ $b
  $s > 50  ⟹  (TimeComplexity, $s / $p) ∈ $b
 end
end;
```

Listing 7: Passport for box NR, with parallel behavior.

```
∀ A  actor A : location.(p), state.(s),
               component.(c), input.(r)
            ↦ state.(s'), observation.(m),
               output.(r');
```

Listing 8: Common actor passport signature.

Every time an passport is *evaluated* during aggregation, the underlying platform will *perform* the computation(s) described by the left side of the actor signature.

The left side of the signature specifies the *location*, i.e. where in the system the computation takes place; the carried *state* from a previous computation, the functional *component* that defines the input-output relationship, and the concrete *input* data on which to compute. The right side specifies the resulting carried state, a set of *actual observations* about the computation, and the concrete *output* data that is produced.

An example basic actor passport is given in Listing 9. This passport specifies that the actor `SimpleDo` carries out a computation and reports an observation event `Done` annotated with the component reference and the concrete time to result.

This passport is sufficient to perform computations, but likely insufficient to analyze components whose behavior is dependent on the input data or the physical resource where the computation takes place. For this, a more suitable passport is given in Listing 10. This passport combines three features. It reports different observation events depending on whether the computation has terminated successfully or has encountered an error. It also binds both the component reference, the input reference, the placement reference and, if successful, the output reference, to the observation, so that the behavior is fully contextualized. Finally, it also reports the specific timestamps where the computation started and finished, instead of the time to result, so as to enable simultaneous comparisons across computations.

Next to actor passports, the virtual hardware also pro-

```
actor SimpleDo : location.(p), state.(s),
                 component.(c), input.(r)
↦ state.(s'), observation.(m), output.(r') =
      (Done, $c, $$TTR) ∈ $m;
```

Listing 9: Example actor passport.

```
actor MaybeFail : location.(p), state.(s),
                  component.(c), input.(r)
↦ state.(s'), observation.(m), output.(r') =
(Good) ∈ $$Result
⟹ (Done, $c, $r, $p, $r', $$T0, $$T1) ∈ $m
(Bad) ∈ $$Result
⟹ (Failed, $c, $r, $p, $$T0, $$T1) ∈ $m;
```

Listing 10: Example actor passport with support for failures.

```
synthesis L : observation.(m) =
provided
  (Done, $e, $r, $p, $r', $t0, $t1) ∈ $m
use
  (LatencyPBPIPL, $e, $r, $p, $t0, $t1−$t0) ∈ $m
end;
```

Listing 12: Example synthesis for per-box, per-location, per-input latency.

vides resource passports to describe the underlying platform. All resource passports are unary and have the following common signature:

```
∀ R  resource R : hwprops.(p) ;
```

Each resource passport further expresses properties that hold in the set provided as input. For illustration, an example resource database is listed in Listing 11. This defines 4 processing units `W1_0`, `W1_1`, `W2`, `W3` over three fully connected network nodes `N1`, `N2`, `N3`. Each processing unit is defined from hardware resources on that node, here shared memories and hardware cores, which are also fully enumerated with their properties.

3.3 Synthesis passports and term derivation

In addition to box, actor and resource passports which assume concrete semantics (agreed by convention) for their vocabularies of terms, a CAL-equipped system can also be extended with *synthesis passports* which define new terms purely symbolically.

For example, we may desire to define a component's *latency* and *throughput* over a set of observation events. We provide an example passport `L` to synthesise latencies in Listing 12. This maps every observation event from `MaybeFail` to a `LatencyPBPIPL` event, which substitutes the end time by the time to result, and drops the reference to the output data. This passport can then be complemented by Listing 13 to synthesise throughput. Here, the throughput is defined as the number of latency events unified from the observation sets and a *request set*. A separate, explicit handle to requests is necessary because throughput can only be defined over a given interval of time. Here we assume that requests also filter for specific components and placement. The resulting throughput unit is the number of activations of the box per unit of time. The unit of time is context-dependent: when `T` is combined with the `MaybeFail` and `L` passports, the unit of time is milliseconds.

To illustrate `L` and `T`, we consider the example set of monitoring events given in Listing 14, generated by applying the `MaybeFail` actor to the image pipeline defined earlier and two input frames.

In this data, every computation event lists the start and end times, here assumed to be expressed in milliseconds after the start time of the entire application. This data also shows that the NR box has executed on the processing unit `W1_0`, whereas Hist has executed on `W2`. We mask the frame and histogram data references behind simple identifiers for clarity; in an actual implementation the events would either specify direct memory pointers or database identifiers to the concrete images.

```
synthesis T : observation.(m), request.(rq) =
provided
 (Request, $e, $p, $tb, $te) ∈ $rq,
 let $s :=
   { (LatencyPBPIPL, $ex, ∗, $px, $t0, $l) ∈ $m
 ‖ $ex = $e ∧ $px = $p ∧ $t0 ≥ $tb ∧ $t0 ≤ $te },
 (CALCount, (LatencyPBPIPL), $n) ∈ $s
use
 (ThroughputPBPL, $e, $p, $tb, $te,
      $n / ($te − $tb)) ∈ $m
end;
```

Listing 13: Example synthesis for per-box, per-location throughput.

```
{ (Done, (NR), ⟨f0⟩, (W1_0), ⟨t0⟩, 12, 24),
  (Done, (NR), ⟨f1⟩, (W1_0), ⟨t1⟩, 25, 39),
  (Done, (Hist), ⟨t0⟩, (W2), ⟨h0⟩, 26, 42),
  (Done, (Hist), ⟨t1⟩, (W2), ⟨h1⟩, 43, 51), } ⊂ m
```

Listing 14: Example monitoring data

When unifying this data via `L`, we derive the extension in Listing 15; with `T` and a single request to investigate component NR over WR_0 and a window of 100ms, we derive the extension in Listing 16. Note how numeric precision is preserved by the use of rational numbers in expressions.

4. RELATED WORK

Auto-tuning of performance of parallel programs is quite different to auto-tuning sequentially running code. Research has been done to translate sequential programs automatically into parallel programs at the level of language constructs such as loops. For example, Williams et al. have developed a framework to parallelise sequential Fortran 95 code for a few numerical application domains [2].

The most related work is a program specialisation framework developed by Kessler et al. [3]. Their approach is based on the resource-aware gray-box composition of components that encapsulate sequential or explicitly parallel code. The approach extends program code with code annotations representing meta-data used to predict the execution time. Different implementation variants come with different performance meta-data. A composition tool is used to transform a set of independent calls into a construct with dynamic dispatch to select at runtime the best performing computation strategy. The choice of the computation strategy is based on statically computed lookup tables for the schedule.

```
{ (LatencyPBPIPL, (NR), ⟨f0⟩, (W1_0), 12, 12),
  (LatencyPBPIPL, (NR), ⟨f1⟩, (W1_0), 25, 14),
  (LatencyPBPIPL, (Hist), ⟨t0⟩, (W2), 26, 16),
  (LatencyPBPIPL, (Hist), ⟨t1⟩, (W2), 43, 8), } ⊂ m
```

Listing 15: Example concrete latency synthesis.

```
// example network descriptions with 3 fully connected nodes
resource N1 : hwprops.(h) = (Neighbour, N2) ∈ $h ∧ (Neighbour, N3) ∈ $h ;
resource N2 : hwprops.(h) = (Neighbour, N1) ∈ $h ∧ (Neighbour, N3) ∈ $h ;
resource N3 : hwprops.(h) = (Neighbour, N1) ∈ $h ∧ (Neighbour, N2) ∈ $h ;
// example memory descriptions; units are in KiB
resource M1 : hwprops.(h) = (Capacity, 1024) ∈ $h ; // 1MiB
resource M2 : hwprops.(h) = (Capacity, 4096) ∈ $h ; // 4MiB
resource M3 : hwprops.(h) = (Capacity, 256) ∈ $h ; // 256KiB
// example hardware core descriptions; frequencies in MHz, cache capacities in KiB
resource C1_0 : hwprops.(h) = (Freq, 1000) ∈ $h ∧ (Cache, 512) ∈ $h ;
resource C1_1 : hwprops.(h) = (Freq, 1000) ∈ $h ∧ (Cache, 512) ∈ $h ;
resource C2 : hwprops.(h) = (Freq, 2000) ∈ $h ∧ (Cache, 128) ∈ $h ;
resource C3 : hwprops.(h) = (Freq, 800) ∈ $h ∧ (Cache, 256) ∈ $h ;
// example descriptions for processing units
resource W1_0 : hwpropos.(h) = (ShMem, M1) ∈ $h ∧ (Core, C1_0) ∈ $h ∧ (Node, N1) ∈ $h ;
resource W1_1 : hwpropos.(h) = (ShMem, M1) ∈ $h ∧ (Core, C1_1) ∈ $h ∧ (Node, N1) ∈ $h ;
resource W2 : hwpropos.(h) = (ShMem, M2) ∈ $h ∧ (Core, C2) ∈ $h ∧ (Node, N2) ∈ $h ;
resource W3 : hwpropos.(h) = (ShMem, M3) ∈ $h ∧ (Core, C3) ∈ $h ∧ (Node, N3) ∈ $h ;
```

Listing 11: Example hardware property database.

```
{ (ThroughputPBPL, (NR), (W1_0), 0, 100, 2/100)
} ⊂ m
```

Listing 16: Example concrete throughput synthesis.

The work presented in this paper based on CAL allows for a more generic component characterisation. Further, the hardware characterisation described by Kessler et al. only considers the number of processors, while our approach provides a richer vocabulary to characterise resources. While Kessler et al. work at the call interface between components, the approach described in this paper works at the coordination level of components with a clear separation between concurrency specification and component implementation. Because of this we deem it to be an interesting experiment to apply our overall approach to other coordination and components models [5, 6].

5. CONCLUSIONS

We have presented a system design that allows us to specify, monitor and relate a wide range of properties from computational components, the coordination glue that allows for concurrent execution of components down to the executing hardware platform. We believe that the presented approach targets one of the main difficulties in assessing the behaviour of a parallel system by allowing users to systematically specify which properties they are interested in and by equipping the system-side components with observation and reporting facilities to collect and provide access to the desired measurements. This is only the first step towards automatic system refinement as an autonomous process without user interaction, however, with analysis, heuristics and probabilistic prediction algorithms in place we believe this to be a reachable goal.

As was mentioned before, this paper describes the current state of an ongoing research project. We are still exploring the set of potentially useful properties on all levels and ways to relate them to each other to provide the most benefits to a user. One important realisation is that it is infeasible to anticipate what users find most useful in a particular (i.e. their own) hardware/software configuration. Hence, we opted to make the observation and synthesis user-definable through CAL so as to empower users to define their own metrics and ways to obtain the most supportive measurements.

The current implementation state of the tools varies. While the S-Net tool chain has been under development for over 5 years and has reached some stability, the observation system and the CAL aggregator are in a prototype stage. Nonetheless, all tools are available for download (open-source) at `http://www.project-advance.eu/`.

6. ACKNOWLEDGEMENTS

This research is supported by the European Union under grant IST-248828 (FP7 ADVANCE).

7. REFERENCES

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

[2] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proc. Int'l Parallel and Distributed Processing Symposium*, pages 1–12, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[3] C. W. Kessler and W. Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

[4] 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*, Philadelphia, PA, USA, Jan. 2012.

[5] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33:709–724, October 2007.

[6] A. Omicini and M. Viroli. Review: coordination models and languages: From parallel computing to self-organisation. *Knowl. Eng. Rev.*, 26:53–59, 2011.

[7] A. Shafarenko and R. Kirner. CAL: A language for aggregating functional and extrafunctional constraints in streaming networks. Technical report, University of Hertfordshire, Hatfield, UK, Jan. 2011. available at `http://arxiv.org/abs/1101.3356`.