A Framework for Characterizing Predictable Platform Templates

Version 2

SEYED HOSEIN ATTARZADEH NIAKI, MARCUS MIKULCAK, FRANCESCO ROBINO, AND INGO SANDER

Technical Report
Stockholm, Sweden 2014

Electronic Systems
School of Information and Communication Technology
KTH Royal Institute of Technology

TRITA-ICT/ECS R 14:01, ISSN 1653-7238, ISRN KTH/ICT/ECS/R-14-01-SE
A Framework for Characterizing Predictable Platform Templates

Seyed Hosein Attarzadeh Niaki, Marcus Mikulcak, Francesco Robino, and Ingo Sander

September 22, 2014

The design of real-time multiprocessor systems is a very costly and time-consuming process due to the need for extensive verification efforts. Generic correct-by-construction system-level design flows, targeting predictable platforms, would help to tackle this problem. Unfortunately, because system-level design problems are formulated monolithically, existing methods are either not powerful enough to perform efficient design space exploration, over-customized to a specific class of platforms, or do not allow to be extended with new heuristics and solving methods, which makes their reuse difficult. We present a formal framework to explicitly capture and characterize predictable platform templates that can be used to formulate a generic design flow for real-time streaming applications in a composable manner. A proof-of-concept implementation of such a flow is performed and used to map a JPEG encoder application onto an FPGA-based time-predictable platform.

1 Introduction

Current standard implementation platforms and design methods for multiprocessor real-time embedded systems cannot guarantee preservation of the critical properties of real-time systems and hence, extensive verification tasks dominate the total cost of system design. Unpredictable access times to shared resources in MPSoCs due to contention and lack of analysis methods for predicting the execution services provided by modern architectures [31] are among the problems which are rooted in the implementation platforms.

Predictable platforms are promoted as a solution which provide timely-guaranteed services to the applications [29]. Once these platforms are used in a system-level design paradigm such as platform-based design (PBD) [17], the design flow can exploit the properties of a formal applications model and the services provided by the platform to explore the design space and achieve an efficient implementation [9, 11]. Platforms are often introduced to the designs flows as a set of components in a structured format,
such as XML [22] or architecture description languages like AADL [7]. Components are configured by parameters and annotated with performance and cost metrics.

However, such formats and languages cannot capture all the information required to characterize predictable platforms explicitly. Instantiation constraints, which are the rules to compose the components as a valid platform, and also the services and costs provided/implied by the platform are not visible in these formats and it is assumed that the design tools have implicit knowledge about them. As a result, tasks such as design space exploration (DSE) are formulated as a monolithic problem which has indistinguishable knowledge about the semantics of the input application model, the type and characteristics of the elements available in the target platform, the mapping and scheduling core, etc. Consequently, current design flows are deployed for a single, or a fixed class of platforms and introducing new target components and platforms is impossible without reformulating the often complex mapping and design space exploration (DSE) problems. Also, the platform characteristics are redundantly repeated in other steps of the flow such as generation of virtual prototypes. Additionally, design tools mix the problem with the way it is solved. Thus, it is not possible to switch between or introduce new heuristics based on the characteristics of the application and type of the target platform.

This paper introduces a declarative framework for formal characterization of predictable platform templates, which captures structural, performance, and cost-based characteristics of the platforms parametrically. The framework enables a) a novel approach to formulate the DSE problem for PBD of real-time MPSoCs in a composable way as a constraint satisfaction problem (CSP); and b) automatic generation of virtual platforms for simulation-based validation of candidate solutions. Based on constraint programming (CP), we illustrate the potential and flexibility of our approach by developing a flow which generates DSE tools, aiming at finding efficient platform instances and mapping and scheduling of applications on it. Different heuristics can be used to solve the same CSP.

To summarize the contributions, we
- introduce a component-based framework for characterizing the services and costs provided by predictable platform templates (Sections 3, 4);
- demonstrate the introduced framework in action by characterizing a set of typical predictable component templates as a predictable platform template;
- automatic generation of DSE problems which correctly implement task data-flow graphs with real-time requirements as constraint satisfaction problems (Section 5);
- realize the DSE problem using CP to implement a JPEG encoder application on top of an FPGA-based predictable platform template (Section 7).

2 Background

A constraint satisfaction problem (CSP) [5] is expressed as a set of constraints which must be satisfied over a set of variables with given finite domains. Formally, a CSP is defined as a triple \((X, D, C)\), with \(X\) being a set of variables where each \(x_i \in X\) can take values
from a domain \( d_i \in D \). Each constraint \( c \in C \) involves some variables \( x_j, x_k, \ldots \in X \) and is a subset of possible values that each variables can take i.e., \( c \subseteq d_j \times d_k \times \ldots \). The helper function \( \text{var}(c) = \{x_j, x_k, \ldots\} \) returns the variables involved in a constraint. Constraints are typically formulated as expressions containing variables of interest, such as \((x_1^2 + 2x_2 \leq 5) \land (x_1 \neq x_2)\). An assignment fixes a value for one or more constraint variables from their domains. A complete assignment is one which assigns values to all the variables and if it does not violate any of the constraints, it is also a solution to the CSP. Assume two CSPs \( P_1 = (X_1, D_1, C_1) \) and \( P_2 = (X_2, D_2, C_2) \). The composition of \( P_1 \) and \( P_2 \), denoted by \( P_1 \times P_2 \) is also a CSP which has the variables \( X_1 \cup X_2 \) and constraints \( C_1 \cup C_2 \).

We use the composability of CSPs to distinguish the subset of the system-design problem which roots from the platform characteristics from the logic of the DSE and capture them using separate frameworks with a well-defined interface. Constraint programming (CP) \[3\] is a declarative programming paradigm which provides languages and tools to express and solve CSPs. CP can be used both for solving satisfiability and optimization problems. An advantage of CP is that it separates the formulation of the problem from the search algorithm used for solving it.

### 3 Predictable Component Templates

A **component template** \( m \) provides a set of services to the application at a given cost. The behavior of \( m \) is tuned by setting a set of **variables** \( V_m \) upon instantiation. Choosing specific values for variables of a component template \( m \) restricts the set of possible values for other variables. An **instantiation constraint set** \( C_m \) captures these restrictions for \( m \). The main purpose of \( m \) is to provide a set of **service functions** \( R_m \) to the application. Each component in an instantiated system comes with its own cost. Depending on the values of a component template variable, the set of component **cost indicators** \( O_m \) provide the design space exploration tool with the costs associated with each component template \( m \).

**Definition 1** A predictable component template is a 4-tuple \( (V, C, R, O) \) where:
- \( V \) is a set of component template variables where each \( v \in V \) is a variable with a finite domain;
- \( C \) is a set of instantiation constraints where each \( c \in C \) has the constraint variables \( \text{var}(c) \subseteq V \);
- \( R \) is a set of service functions where for each \( r \in R \) and \( n \in \mathbb{N} \) arguments, \( r : (V \cup V_{\text{ext}})^n \rightarrow D_F \); and
- \( O \) is a set of cost indicators where for each \( o \in O \) and \( n \in \mathbb{N} \) arguments, \( o : (V \cup V_{\text{ext}})^n \rightarrow D_F \).

\( D_F \) is a finite domain set. \( V_{\text{ext}} \) is a (possibly empty) set of extra variables which are defined in a CSP interface (to be detailed later). The helper functions \( \text{vars}(m) = V \), \( \text{servs}(m) = R \), and \( \text{costs}(m) = O \) are defined for convenience.
3.1 Component Allocation and Instantiation

An allocation of a component template \( m \), denoted by \( A_m \), is a copy of its variable set. If \( m \) is allocated \( n \) times, then \( A_m = \{ A_{m,1}, \ldots, A_{m,n} \} \) is called the allocation set of \( m \). An instance of a component template \( m \), denoted by \( I_m \), is a complete assignment to one of its allocations while satisfying all of its instantiation constraints. If \( m \) is instantiated \( n \) times, then \( I_m = \{ I_{m,1}, \ldots, I_{m,n} \} \) is the instantiation set of \( m \). The helper functions \( \text{allocs}(m) = A_m \) and \( \text{insts}(m) = I_m \) are defined for convenience.

3.2 Component Template Examples

Components of an embedded platform can be categorized roughly as processors, interconnections, storage, and IO-elements. We show how example models of two of these categories are characterized using our formalism.

3.2.1 Processors

No matter how complex a processor is from the micro-architecture perspective, from a system-level point of view, the guaranteed services that a predictable processor provides for execution of software tasks is expressed in terms of Worst/Best-Case Execution Times (WCET/BCET). A processing element which is a candidate for running a set of computation tasks \( K \) is characterized as a processing component template \( m_{pe} = (V_{pe}, \emptyset, R_{pe}, O_{pe}) \) where for \( \forall k \in K \) we have \( V_{pe} = \{ v_{freq} \} \), \( R_{pe} = \{ r_{wcet_k} \} \), and \( O_{pe} = \{ o_{area}, o_{mem_k} \} \). The frequency of the processor \( v_{freq} \in \{ f_1, f_2, \ldots, f_n \} \) is a component variable. The processor provides a WCET service \( r_{wcet_k} = WCEC_{pe}(k) \times \frac{1}{v_{freq}} \) where \( WCEC(k) \) is the worst-case execution cycles for computation task \( k \), derived from the analysis tool. The constant area cost indicator \( o_{area} = pe_{area} \) and the worst-case memory requirement cost indicator \( o_{mem_k} = WCMR(k) \) of each application tasks complete the characterization of our example.

3.2.2 Interconnection networks

Typically, a time-predictable interconnection network must transfer data among different cores with a guaranteed throughput and/or latency. As an example, a simple TDM bus which is supposed to implement a set of communication tasks \( \kappa \in \mathcal{K} \) can be characterized as a communication component template \( m_{bus} = (V_{bus}, C_{bus}, R_{bus}, O_{bus}) \) where \( V_{bus} = \{ v_{ports}, v_{tbl}, v_{freq} \} \), \( C_{bus} = \{ c_{ports tbl}, c_{tbl} \} \), \( R_{bus} = \{ r_{wcet} \} \), and \( O_{bus} = \{ o_{area} \} \) in which the component variables are the frequency of the bus \( v_{freq} \in \{ f_1, f_2, \ldots, f_n \} \) as the inverse of the length of each slot in terms of global time units, the number of IP blocks connected to the bus \( v_{ports} \in \mathbb{N} \), and the TDM table \( v_{tbl} = \{ v_{tbl,i} \in \mathbb{N} \mid i = 1..|v_{tbl}| \} \) which is an array indicating how many time slots each dedicated to each component connected to the bus. In addition,

\[
\begin{align*}
\text{c}_{ports tbl} & : v_{tbl,i} = 0; \quad \forall i > \text{ports} \\
\text{r}_{wcet} & = \sum_{\text{v}_{bl}} v_{bl} \div v_{bl,src(k)} \times \frac{1}{v_{freq}}; \quad \forall \kappa \in \mathcal{K} \\
o_{area} & = v_{ports} \times (|v_{tbl}| \times \text{mem area} + \text{port area})
\end{align*}
\]
A single instantiated constraint fixes the unused elements of the TDM table. The worst-case communication time per word provided by this communication component template is expressed in terms of the number of associated time-slots that a source has access to the bus. src(κ) here denotes the component initiating the κ transaction. The chip area used by the bus is the cost indicator in this example where for each port the amount of logic required for connecting an IP block is assumed to be a constant port_area and each memory location for the local TDM tables take mem_area units.

4 Predictable Platform Templates

A platform template m introduces a set of template components, their dependencies on each other, services provided by the platform, and overall system costs.

In an instance of a platform template (to be defined later), the values assigned to variables and the number of instances of different components might be interdependent. We capture this fact using a set of interdependent instantiation constraints C. A platform template m can provide a set of overall cost indicators O that compute the overall system costs based on the cost indicators of the instantiated component templates. For example, the overall area usage of the above MPSoC is the sum of the area usage of its individual components plus some additional glue logic.

Definition 2 A predictable platform template m is a triple (M, C, O) where:
• M is a set of predictable component templates;
• C is a set of interdependent instantiation constraints where for each c ∈ C we have var(c) ⊆ \bigcup_{m \in M} allocs(m);
• O is a set of global system cost indicators where for each o ∈ O we have o : (\bigcup_{m \in M} allocs(m))^n \rightarrow DF; \ n \in \mathbb{N}.

4.1 Platform Template Instantiation

A platform template is instantiated where at least one of its component templates are allocated and the instantiation set of all of its component templates satisfies the inter-dependent instantiation constraints.

4.2 The Platform Template CSP

Assume a platform template m = (M, C, O). The constraint satisfaction problem associated with m is Pm = (Xm, Dm, Cm) where Xm contains all the variables of allocated components templates plus the extra variables from the interface. The solutions to Pm is the set of platform instances that satisfy all the constraints.

4.3 A Platform Template Example

Based on the component templates introduced in Section 3.2, we can characterize a predictable platform template mp = (Mp, Cp, Op) with Mp = \{mpe, mbus, msmp\}, Cp =
\{\tau_{bus,smp}, \tau_{bus,pe}, \tau_{smp,pe}\}, \text{ and } \overline{O}_p = \{o_{\text{area}}\} \text{ as}

\tau_{bus,smp} : (|A_{m_{bus}}| = 0) \lor (|A_{m_{smp}}| = 0)
\tau_{bus,pe} : A_{m_{bus,1}} \cdot v_{\text{ports}} = |A_{m_{pe}}|
\tau_{smp,ip} : |A_{m_{smp}}| + 1 = |A_{m_{pe}}|
\overline{o}_{\text{area}} = \sum_{m \in M_p} \sum_{A \in A_m} A \circ o_{\text{area}}

The circle operator used to access the elements of an allocated component, e.g., \(A_{m,i} \circ v\) refers to variable \(v\) of the \(i\)-th allocation of component template \(m\). In the above platform template, the \(\tau_{bus,smp}\) restricts the interconnection used in the platform to be either bus-based or composed of serial connections using simplex links. In the first case, \(\tau_{bus,ip}\) ensures that the number of IP-blocks connected to the bus is equal to the number of bus ports. In serial simplex-based communication, the \(\tau_{smp,ip}\) constraint establishes a relation among the number of IP blocks and communication links. The overall cost indicator \(\overline{o}_{\text{area}}\) sums up the individual area cost of the component allocations in the platform.

### 4.4 Representation

The platform characterization is captured as an XML+mzn format. The XML part includes all the characterization elements except the implementation of the constraints, including the service and cost functions. The mzn file captures the constraints as predicates of the CP language Minizinc [20]. Figure 1 shows this representation for the above example where the XML part and the implementation of the \(\tau_{wcct}, o_{\text{area}}, \text{ and } \tau_{smp,ip}\) predicates are shown.

### 5 A Generic DSE Tool for Predictable Platforms

As the first application of the suggested framework, we introduce an example of a basic design-space exploration (DSE) tool which maps a given input application onto an instance of a platform template characterized using the presented approach. The design flow is shown in Figure 2. The task (data-flow) graph model is used as the input application model which distinguishes between computation task nodes, which demand WCET services and communication nodes, which require WCCT services. The application is going to be mapped to a characterized platform template such as the one presented in Section 4. The designer might also provide additional real-time constraints such as maximum end-to-end delay, or cost constraints such as total area, total memory, etc. The mapping and scheduling problem for finding a solution is formulated as a constraint satisfaction problem (CSP). Once this problem is composed with a platform CSP, a DSE tool for allocation, mapping, and scheduling of the input application targeting the respective platform is generated. After solving the CSP using a finite-domain solver, platform and application synthesis is performed to realize the final implementation.
<platform name="simple">
    <component name="pe">
        <comp_parameter name="cycle" domain="{10,20,40}" />
        <comp_service name="wcet" />
        <application_arg name="mapped_task" domain="tasks" />
        <comp_cost cost="area" />
    </component>
    <component name="smp">
        <comp_parameter name="cycle" domain="{10,20,40}" />
        <comp_service name="wcct" />
        <comp_cost cost="area" />
        <comp_cost cost="mem" />
    </component>
    <component name="bus">
        <comp_service name="wcct" />
        <comp_cost cost="area" />
    </component>
    <plat_inst_cons constraint="bus_or_smp" />
    <plat_inst_cons constraint="bus_ip" />
    <plat_inst_cons constraint="smp_ip" />
    <plat_service name="pe_wcet" />
    <comp_cost cost="area" />
</platform>

... predicate smp_wcct(var int: wcct, var int: cycle, var int: depth) =
    wcct = 1 * cycle;
predicate smp_area(var int: area, var int: cycle, var int: depth) =
    area = 100 + depth * 10;

... predicate smp_ip =

Figure 1: The predictable platform template captured as an XML file capturing the framework elements and an mzn file including the implementation of the predicates (partly represented here).
Figure 2: Example of a basic design flow based on the presented approach. The green rectangles denote models and the blue ones the activities. The filled activities are automated. By using our proposed characterization framework, the target platform template can be replaced without modifying the design flow, and the DSE models and TLM virtual prototypes can be generated automatically.

### 5.1 Decision Variables

An application is modeled using an acyclic task graph \((K, E)\) in which the nodes \(K = K \cup \mathcal{K}\) present computation and communication tasks and the edges \(E\) capture the causal dependencies between them. The execution of tasks is assumed to be atomic and the respective communications happen only before and after its execution. Each communication task \(\kappa_i\) in the graph is annotated with a token size \(s_i\) which is communicated by it. Each task \(k_i \in \mathcal{K}\) is a triple of decision variables \((\delta_i, l_i, \rho_i)\), where \(\delta_i\) is the time \(k_i\) is scheduled to start, \(l_i\) is the WCET (WCCT) of it and \(\rho_i\) is the component onto which it is mapped. Thus, the application variables are:

\[
X_{\text{app}} = \bigcup_{k \in \mathcal{K}} \{\delta_k, l_k, \rho_k\}.
\]

An array of lookup variables \(\ell\) of length \(|M| \times \max(|\text{allocs}(m_i)|)\) shall contain the values for the WCET and WCCT services provided by different instances of the available components templates. Thus, for mapping variables \(X_{\text{map}} = \ell\).

### 5.2 Constraints

Every edge \(e_{i,j}\) represents a dependency between two tasks \(k_i\) and \(k_j\) which implies a causal precedence among the tasks. These application modeling constraints can be expressed as:

\[
C_{\text{app}} = \{(\delta_i + l_i \leq \delta_j) \mid \forall e_{i,j} \in E\}
\]

which effectively captures the precedence between the computation and computation tasks. For mapping constraints, first there cannot be more than a single task running on each processor at any point in
time. As described in [18], instead of posting $n(n-1)/2$ disjunctive constraints, one can represent each computation node as a rectangle in the time/resource space with coordinates $((\delta_i, \rho_i), (\delta_i, \rho_i + 1), (\delta_i + l_i, \rho_i), (\delta_i + l_i, \rho_i + 1))$ and use the `diffn` constraint on all $k_i \in K$. This constraint takes the vectors $x, y$ as the origins of rectangles and also $\Delta x, \Delta y$ as their sizes and ensures that the rectangles do not overlap in at least one dimension. Thus, $C_{\text{map}, 1} = \text{diffn}(\delta, \rho, l, \langle 1, 1, \ldots, 1 \rangle)$ for all computation tasks. The WCET (WCCT) of a task $l_i$ depends on which resource it is mapped to, represented by $\rho_i$. The `element` constraint is used to relate these two sets of variables. `element`(i, x, y) simply constrains y to be the result of lookup of the $i$th element of vector $x$, i.e., $y = x_i$. Thus, $C_{\text{map}, 2} = \{\text{element}(\rho_i, l_i) \mid i \in K\}$. An additional set of constraints $C_{\text{map}, 3}$ is posted to populate the lookup tables in such a way that for $l_i$ where $i = j \times \max(|\text{allocs}(m_i)|) + k$, represents the WCET (WCCT) service provided by the $k^{th}$ allocation of the $i^{th}$ component.

5.3 Additional Constraints

More detailed constraints such as the maximum value of an overall cost (e.g., area), the minimum throughput, or mapping constraints based on user experience to aid the tool to find the solutions faster can be posted by the designer. For example a real-time constraint which may provide a lower bound on the minimum intended throughput of the system could be formulated as: $C_{\text{ext}} = \{\max(\delta_k + l_k) < \frac{1}{\text{min., throughput}}\}; \ \forall k \in K$.

5.4 The Design Space Exploration Problem

The generic DSE problem is simply a constraint satisfaction problem (CSP) in form of $P_D = (X_D, D_D, C_D)$ where $X_D = X_{\text{app}} \cup X_{\text{map}}$ and $C_D = C_{\text{app}} \cup C_{\text{map}, i} \cup C_{\text{ext}}$. Having the CSP formulation for a generic DSE $P_D$ and the CSP problem for an example of a predictable platform template $P_m$, we can generate the a DSE problem specialized for our platform, in turn as a CSP, by $P_D \times P_m$.

5.5 Implementation

A script reads the characterized platform template in the XML+mzn format and based on a template file which defines the MiniZinc formulation of the generic DSE tool generates another MiniZinc script which implements the specialized DSE problem for the platform $P_D \times P_m$. A generic constraint programming (CP) solver can be used to solve this problem. Problems formulated as CP languages such as MiniZinc can be used by a variety of solver types such as finite domain and linear programming tools. The Gecode [8] solver is used as a backend to solve the DSE problem. CP tools typically allow the user to specify how the solution space should be explored. For example, Gecode as a finite domain solver can be extended by programming new propagators and branchers. Providing support for advanced features such as parallel and restart-based search makes Gecode a powerful CP solver. Other methods such as hybrid solvers can be used to solve the problem even more efficiently.
6 Automatic Generation of Virtual Prototypes

Very often, it is not sufficient to perform only static DSE based on analytical methods for system design. Imprecise characterization of the platforms, being able to tolerate occasional deadline misses of soft real-time tasks for the sake of more efficient solutions, and employing analysis methods which require execution trace of the system are among the reasons where dynamic evaluation of the candidate solutions becomes necessary. This leads to a hybrid flow where an analytical DSE phase is succeeded by dynamic evaluation of candidate solutions [16]. Virtual prototypes (VPs) [10] are fast simulation models of the systems used to mimic the execution of applications on target platforms. However, developing VP models is non-trivial and time-consuming and for the purpose of DSE, they even need to be generated automatically. Using the characterized component services captured in our proposed framework, it is possible to automatically generate loosely timed TLM 2.0 VPs for each instance of the platform with the application mapped and scheduled onto them [13]. This is depicted in Figure 2. In this approach, generic virtual processors are used to execute the mapped tasks which are annotated the WCET services provided to them. The tagged sockets from the interoperability layer of TLM 2.0 are used to generate generic communication medium models which annotate the transactions with respective characterized WCCT services. In this way, no additional library of simulation models need to be provided for each target platform. These high-level models can be used to get early simulation traces and also for gradual refinement using more accurate component models. More details are found in [13].

7 Case Study

We have characterized two different FPGA-based predictable platform templates using the presented approach presented in Sections 3 and 4, and used the flow presented in Section 5 to generated DSE problem models for each platform and implement a JPEG encoder application on top of them.

7.1 The psopc Platform

The Altera System on Programmable Chip (SOPC) is modified for more predictability by utilizing TDM arbitration on the slave ports and processing elements consisting of a Nios II/e processor with local instruction and data memories with fixed access time. Extended features can be easily supported if there are WCET analysis methods available. A software layer provides FIFO-based message-passing communication between processes on top of dedicated shared memories. This platform is characterized as a special case of the example presented in Section 4.3 excluding the simplex link. The area cost $o_{area}$ of the processing element is 3 650 logic elements (LE), and $port_{area}$ and $mem_{area}$ are 100 LE and 10 LE, respectively.
7.2 The nocgen Platform

The NoC system generator of [23] constructs platforms consisting of processing elements similar to the one in psopc, excluding the DMA unit, and a network-on-chip which includes a routing switch and a resource network interface (RNI) per tile. Communication of data tokens between the processes is facilitated by a message-passing software layer. The area cost of an RNI is 220 LE and for the corner and normal switches it is 360 LE and 1100 LE respectively. This platform is not designed for predictability but the WCCT formula for communicating each token is based on the worst-case latency for the best effort traffic [15] as \( 4 \times \frac{1}{v_{freq}} \times (8 + 40 \times v_{dim}) \), for \( v_{dim} \times v_{dim} \) NoCs.

7.3 The Application

A Homogeneous Synchronous Dataflow (HSDF) graph model of the JPEG compression algorithm is captured in ForSyDe-SystemC [2], functionally verified by simulation, and introspected to generate an intermediate representation as a task graph with accompanying C code of the tasks. The model consists of processes for reading the bitmap (RB) as blocks of 16×16 pixels, color conversion and sub-sampling (CC), discrete cosine transformation (DCT) and Huffman coding (H) for four luminance and 2 chroma samples, and finally concatenating steps (CS) and writing the final JPEG (WJ). The performance of each task is determined by measurement-based WCET analysis for the Nios II/e core.

7.4 Results

The generated DSE problems can face the combinatorial explosion problem and appropriate search strategies investigated in the CP community need to be adapted for solving them. For our experiments we a) branch on instantiation, mapping and scheduling variables in order; b) enable a restart-based search with a luby sequence with a scale value of 100; c) enable the parallel search of the solver engine using 8 threads. In addition we add an additional constraint to impose an order on the mapping variables of the DCT tasks to break the symmetry of the application and reduce the design space. In the first step, the DSE generator is invoked to create a mapping and scheduling tool for implementing the application on the psopc platform. By specifying a maximum acceptable area budget of our design and optimizing for iteration length of the task graph, the tool finds a 4-node instance of the platform with the TDM table of the bus together with the mapping and scheduling of the application tasks within 7 seconds. Next, using the same application model and mapping and scheduling logic, a new DSE problem is generated this time for the nocgen platform. Under the same criteria for the previous experiment for maximizing the throughput, the result is again a 4-node solution generated in less than 2 seconds. Figure 3 depicts the mapping of the application on the instantiated nocgen platform. The application is implemented on the suggested instances of both platforms. The measured values for area cost and iteration length of the applications are shown in Table 1 where in every case they are within the predicted range.
Table 1: The predicted and measured worst-case iteration time and area usage of JPEG encoder for the two platforms.

<table>
<thead>
<tr>
<th></th>
<th>psope Predicted</th>
<th>Measured</th>
<th>nocodegen Predicted</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res. usage (LE)</td>
<td>15 000</td>
<td>14 873</td>
<td>5 120</td>
<td>5 108</td>
</tr>
<tr>
<td>Iter. time (µs)</td>
<td>226 044</td>
<td>210 915</td>
<td>226 253</td>
<td>173 131</td>
</tr>
</tbody>
</table>

8 Related Work

Henzinger [12] argues that to build time-deterministic embedded systems from possibly non-time-deterministic components, the key question is how to build adequate deterministic abstractions layers for the embedded system designer and tools. Our framework is a step towards providing such a layer which exposes predictable services provided by a platform without depending on the implementation details. Schoeberl [25] states that although predictability is interesting, it is still the WCET bounds of tasks which is usable for comparison of different architectures. Inline with this, we capture the component services such as WCET and WCCT and use them to decide on allocation and mapping of the input application to the target platform. Thiele and Wilhelm [29] relate time-predictability to the differences between best case and lower bound on the one hand and upper bound and worst case for the execution time on the other. They suggest that we can a) reduce the time-sensitivity to the information not-available at design time, e.g., using a TDMA protocol; and b) match implementation concept with analysis methods, to improve the predictability of the systems; in several layers of hardware,
software, task-level, and even between the layers. Berg and others [4] proposes a set of basic requirements for designing predictable processors. Åkesson and others [1] introduce five generic techniques to achieve predictability and/or composability. The JOP processor [26], the multi-threaded version of the TriCore processor used in the MERASA project [30], the PRET architecture [19], and the CompSOC platform [1] design their architecture carefully in order to combine high performance with predictable and repeatable timings for software execution. Our work provides a platform-based design approach which is applied to the design of systems in all the layers above the hardware architecture, particularly in task-level and distributed layers.

A survey on design flows for system-level design of electronic systems is presented in [9,27]. With respect to the platform-based design (PBD) classification provided in [6], our framework fits into the platform bin, but also acts an interface for integration into a full PBD framework. In approaches such as the Distributed Operation Layer (DOL) [28] and Daedalus [22], the design flow maps parallel applications onto an abstract specification of the platform which contains the structural, performance, and parametric data by performing system-level performance analysis and multi-objective algorithm-architecture mapping. Unlike the way the platforms are characterized in these frameworks, where the mapping and DSE tool must include an implicit knowledge about the meaning and type of the service associated with the predefined abstract components of the platform, our framework presents a more formal and flexible approach to present the above information. An extension to DOL [14] follows the X-chart approach [9] to generate performance analysis models together with the solutions of the DSE. In contrast, because the performance metrics are embedded as service functions during platform characterization in our approach, they can be used during the DSE to generate more optimal solutions. Formulation of the DSE as special cases of CSP such as ILP and satisfiability problems has also been investigated [21,24], where compared to our approach allow more restricted types of constraints and have dedicated solvers. Kuchcinski [18] formulates high-level synthesis and system-level design problem of electronic systems as a CSP. However, the CSP is modeled as a monolithic problem without exploiting the composability of CSPs to distinguish between the platform characterization and the core logic of the mapping and DSE. Our approach follows a more expressive approach to define the structural, performance, and cost-related characteristics of the platform, and can potentially be improved by dedicated heuristics for more efficient solving.

9 Conclusion

We have presented a formal framework for characterizing predictable platform templates which expresses the composition rules of platform templates as instantiation constraints, their provided timely-guaranteed services as service functions, and their overheads as cost indicators. The framework enables a novel approach for design of reliable real-time multi-processor systems by separating the logic of platform characterization from the analysis and synthesis methods. Using our framework, flexible and well-defined interfaces can be introduced between the target platforms and the rest of the design flow
which enables formulation and generation of generic design exploration (DSE) problems in a declarative, composable, and heuristics-extensible style. A classical system-level DSE tool has been adapted to benefit from the introduced framework and is used to map a JPEG encoder example to an FPGA-based predictable platform template.

We can extend our approach further by separating the semantics of the input application from the mapping and scheduling logic in the DSE problem. A more restrictive formulation of service functions can be enforced to enable the application of specific analysis models such as real-time calculus, schedulability analysis, etc. Investigating different solver heuristics and hybrid solvers are interesting to consider also.

References


