Case Study: Constraint Programming in a System Level Synthesis Framework

Shuo Li and Ahmed Hemani

Department of Electronic Systems
School of Information and Communication Technology
Royal Institute of Technology
Isafjördsgatan 39, 16440, Stockholm, Sweden
{shuol,hemani}@kth.se

Abstract. This article presents a case study of using a constraint programming solver in a system level synthesis framework called SYLVA. The solver is used to find the repetition vector of a synchronous data flow graph and serving as the design space exploration engine, which rapidly finds qualified system implementations by solving a constraint satisfaction optimization problem. Each system implementation is a combination of a number of function implementation instances and their cycle accurate execution schedules. The problem to be solved is automatically generated based on the user inputs: 1) a system model to be synthesized, 2) a library containing all the usable function implementations, 3) the performance/cost constraints, and 4) the optimization objectives. Use of constraints programming technique enabled a low cost development of design space exploration engine in addition to gaining ease of use.

Keywords: System Level Synthesis, Design Space Exploration, Constraint Programming.

1 System Level Architectural Synthesis (SYLVA)

System level hardware synthesis is an evolutionary next step after the high-level synthesis. It synthesizes abstract Digital Signal Processing (DSP) sub-systems like modems and codecs, e.g. WLAN, LTE mode, etc. modeled as Synchronous Data Flow (SDF) graphs [1] in terms of pre-characterized Function Implementations (FIMPs). SYLVA [2] is a system level hardware synthesis framework under development in our group. Currently, SYLVA only supports acyclic SDF. It explores the design space in terms of a) number and type of FIMPs, b) number of buffers for each FIMP, and c) pipeline parallelism among them. It also automatically generates the global interconnect and control logic to glue the FIMPs and buffers together into a working system. The design flow based on SYLVA has four steps of which the first two are the focus of this paper. 1) SDF to HSDF conversion 2) Design Space Exploration (DSE) 3) Global interconnect and control synthesis, and 4) Code generation

The key difference with the existing work on CP methods for scheduling tasks on processors, e.g. [3], is that of hardware synthesis vs. software compilation.

B. O'Sullivan (Ed.): CP 2014, LNCS 8656, pp. 846–861, 2014.
© Springer International Publishing Switzerland 2014

Processors can host multiple tasks that are scheduled. In the hardware synthesis case (SYLVA), a FIMP (corresponding to a processor) is a dedicated hardware implementation for a specific function like FFT and only one instance of it can be executed at a time. The scheduling in the context of SYLVA is the relative ordering of the HSDF nodes that are mapped to FIMP instances. The relative order decides the number of FIMPs but not the type. The type selection has been formulated as a CP problem.

Another key difference is that in software compilation, the critical path is known because the arithmetic parallelism of each task is fixed because the processor hardware is fixed. Whereas in case of SYLVA, critical path is not fixed. It depends on the FIMP types that the CP tool evaluates as part of the design space exploration. FIMP types vary in the degree of arithmetic parallelism and can change the total number of cycles that a particular path takes.

1.1 SDF to HSDF Conversion

In this step, the system modeled as SDF graph is converted into a Homogeneous SDF (HSDF) graph [1].

An SDF graph is a directed graph $S = \{A, E\}$. Each vertex is called an actor $a \in A$ that represents a DSP function. The number of data tokens produced or consumed by each actor on each invocation is specified a priori. Each communicational edge $e \in E$ represents a data dependency between two actors. By this definition, two communicating actors a_s (the source) and a_d (the destination) may have different producing and consuming data rates (data token per invocation). Therefore, a_s and a_d have to be invoked at different frequencies to match the data rate of each other. An example SDF graph with four actors is shown in Fig. 1(a), where a, b, c, and d are names of four DSP functions and the data rates are not matched at all.

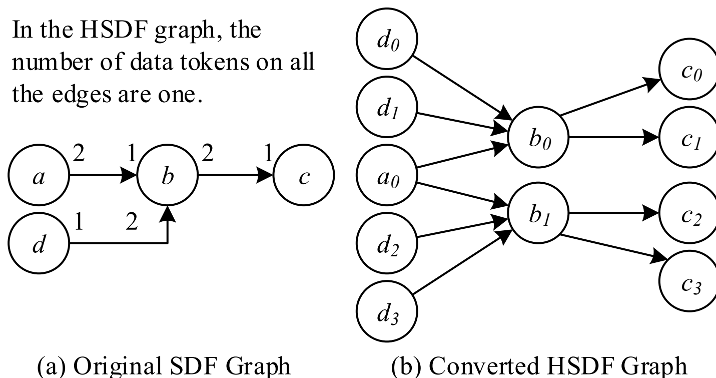


Fig. 1. SDF to HSDF Conversion

An HSDF graph in this article is an SDF graph that all the data rates for all data producer and consumer actor pairs are matched as formulated in 1, where s_e and d_e are the source and destination actors for edge e, respectively.

$$s_e = d_e \quad \forall e \in E \tag{1}$$

Each HSDF actor represents an execution of a DSP function, while each SDF actor represents a DSP function. And the SDF to HSDF conversion will increase the number of nodes. For example, the SDF graph shown in Fig. 1(a) can be converted into the HSDF graph with 11 actors shown in Fig. 1(b), where a_0 is an execution of a, $\{b_0,\ b_1\}$ are two executions of b, $\{c_0,\ c_1,\ c_2,\ c_3\}$ are four executions of c and, $\{d_0,\ d_1,\ d_2,\ d_3\}$ are four executions of d.

SDF to HSDF Conversion can be done by finding the null space of the topology matrix [1]. In SYLVA, we compute the null space by solving a simple Constraint Satisfaction Problem (CSP). The number of variables equals to the number of actors ($|A|$) in the original SDF graph. The possible values of each variable are set to integers from 1 to $2^{32} - 1$. Although the number of possible values is large, experience shows that only a few branches are enough to get the result. For all the experiments in this article, one branch is enough to get the number of SDF actor repetitions and the solver runtime is similar to computing the null space using C# program. The details of the CSP modeling and solving can be found in section 3.

1.2 Design Space Exploration (DSE)

After the first step, we have a number of function executions (HSDF actors) to be implemented by hardware (Function Implementations - FIMPs, which will be explained later in this subsection). The second step is to find the optimal system implementation (solution) in a reasonable time. Each solution specifies the type and number of FIMPs, the number of buffers for each FIMP, and the cycle accurate schedules for them.

The design space can be quite large, since for each HSDF actor, we need to determine following parameters: 1) the FIMP instance to execute it (multiple HSDF actors may share the same FIMP instance in a time-multiplexing fashion) 2) the buffer usage (if each function execution has its own output buffer or not), and 3) the cycle accurate execution schedule (when to start function execution to achieve the desired parallelism) The total number of solutions is the size of the design space $|D|$ and it can be calculated by 2.

$$|D| = \prod_{i=0}^{|A|-1} (2M_i \cdot (T_{max} - t_i \cdot f_i)). \tag{2}$$

$|A|$ is the number of SDF actor. M_i is the number of possible FIMPs for the ith SDF actor (a_i). The term $2M_i$ represents the decision of having output buffer for each HSDF actor or each FIMP instance. T_{max} is the maximum system latency

constraint and t_i is a horizontal vector that contains the execution times for all the FIMPs for a_i. The jth element in t_i is the execution time of the jth FIMP for a_i. f_i is a vertical vector that represents the FIMP assignment decision. The values of each element in f_i can be only 1 or 0 and only one element equals 1 ($\forall i \quad f_i \in \{0,1\}$ and $\sum f_i = 1$). The jth element represents the decision of using the jth FIMP (1) or not (0) to implement a_i. The term $T_{max} - t_i \cdot f_i$ represents the number of possible schedules. In most of the cases, we have more freedom on the schedule side and less freedom on the FIMP type selection side.

If $|A| = 10$, $M_i = 5$ for all actors, $T_{max} = 100$ and all FIMPs have the same execution time of 10 clock cycles ($\forall i \quad t_i \cdot f_i \equiv 10$), the total number of solutions $|D|$ will be $(2 * 5 * (100 - 10))^{10} = 900^{10} = 3.49 \times 10^{29}$. In this example, the schedule term $100 - 10$ contributes much more than the FIMP type selection term $2 * 5$. The Constraint Satisfaction Optimization Problem (CSOP) model will be explained in detail in section 3.

FIMP represents an implementation of a DSP function in terms of three views: *interface view*, *execution view* and *implementation view*. In the rest of this article, a FIMP in a library is called a *FIMP* while a FIMP instantiated for executing one or more HSDF actors is called a *FIMP instance*.

The interface view provides the unique function name, and the input/output data structure of the FIMP. The unique function name defines the name of the DSP function that can be executed by the FIMP. For example, if a FIMP has name = FFT_64, this FIMP can only be used to execute 64-point FFT function, i.e. the HSDF actors derived from the SDF actor whose name = FFT_64. The input and output data structure provides the number of producing/consuming data tokens on each invocation. Three data structure examples with different degrees of parallelism are shown in Fig. 2. All of them has 64 data tokens. Each data token is a 32-bit complex numbers (16-bit fixed point for real and imaginary components). In Fig. 2(a), all 64 numbers are input simultaneously (fully parallel). In Fig. 2(b), all 64 numbers are input one by one (fully sequential). In Fig. 2(c), all 64 numbers are input four by four (partially parallel). At present, two communicating FIMP instances, source and destination, should share same data structure or an explicit data structure conversion should be applied.

The execution view provides a) the function execution timing model (Fig. 3), b) the energy consumption in nJ (nanojoule), and c) the area usage in equivalent gate count. The timing model and the energy consumption are for single execution. In Fig. 3, t_s is the FIMP start time. t_{ie} is the input end time. All input data should be read up by the FIMP till t_{ie}. t_{os} is the output start time. From t_{os}, the FIMP starts sending the output data. t_e is the FIMP end time. Everything is done for one execution till t_e. t_{ie}, t_{os} and t_e are relative times referring to t_s. All of these times are in number of clock cycles and used in the CSOP model.

The implementation view provides a) the implementation style (e.g. ASIC, FPGA, or a specified CGRA) and b) the implementation template (e.g. VHDL code, CGRA configware) for the computation and the output buffer.

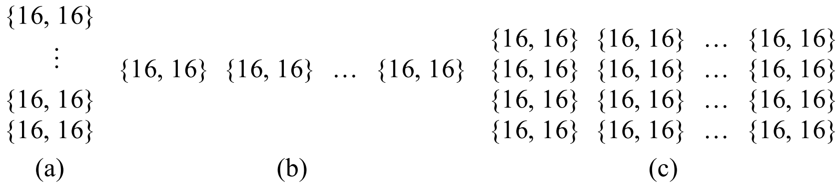


Fig. 2. Data Structure Examples

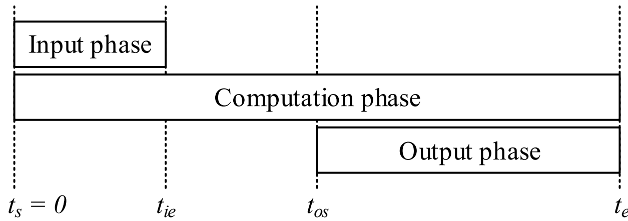


Fig. 3. Function Execution Timing Model

For example, the implementation view of an 64-point FFT can be either a computation core with SRAM in VHDL for ASIC or a MATLAB function for a CGRA to be compiled by the CGRA specific compiler to a configware. This view is not involved in DSE.

Cycle accurate schedule defines the execution schedule of an HSDF actor on a FIMP instance. It consists of a function execution timing model (in terms of t_s, t_{ie}, t_{os} and t_e) and an output buffer end time t_{be} (when the output buffer is released by the data consumer HSDF actor). For example, the HSDF in Fig. 1(b) can have the cycle accurate schedule shown in Fig. 4(a) and (b).

Fig. 4 also illustrates the influence of the output buffer on the FIMP execution schedule. In Fig. 4(a), the FIMP instance B_0 has only one data buffer. Therefore, the second execution of B_0, which is HSDF actor b_1, cannot start until the output buffer of B_0 is released by FIMP instance C_0 and C_1 (HSDF actors c_0 and c_1). The overall system latency is 22 clock cycles. In Fig. 4(b), each of the HSDF actors that are executed by FIMP instance B_0 has its own data buffer. In this case, b_1 can start directly after b_0. The overall system latency becomes 20 clock cycles. The decision of having output buffer for each FIMP instance or each HSDF actor is made be solving the DSE CSOP problem.

1.3 Other Steps

In these steps, the FIMP interconnection and control logics are automatically generated. and the final system implementation is generated. The details can be found in [4] and [5].

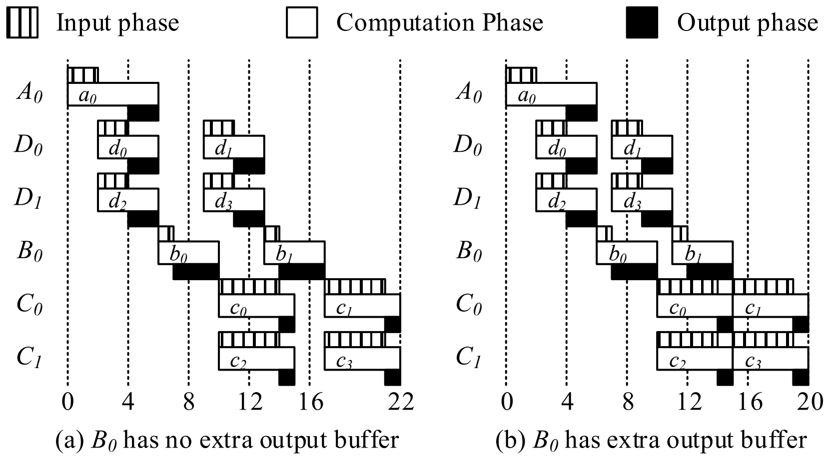


Fig. 4. Cycle Accurate Schedule Example

2 Why Choose Constraint Programming

SYLVA is a research project in which new ideas are constantly emerging and we need a system which enables a high level formal modeling of the problem with minimal effort. This led us to choose the Constraints Programming (CP) framework. Specifically, we list three main motivations:

1) CP has much lower modeling complexity compared with other approaches.
2) CP is suitable for solving scheduling problems, which is critical for us.
3) CP is easy to integrate with other components (e.g. code generator).

2.1 Low Modeling Complexity

CP model supports logical constraints (e.g. logical AND and logical OR) and a full range of arithmetic expressions such as minimum, maximum, or an expression which indexes an array of values by a decision variable. By using CP, constraints can be easily ported from the specification in human language to the solver supported format. In contrast, if we use another approach (e.g. integer linear programming, simulated annealing, genetic algorithm or any other evolutionary algorithms) we need to model the problem in a more complex manner and also define the searching strategy in detail. For example, if integer linear programming is used, we need to formulate the constraints by using linear equations or inequalities, and we cannot use a variable to index another variable or perform logical operations. If an evolutionary algorithm is used, we need to model the problem as well as define the searching strategy by assigning parameters for searching (e.g. randomness simulated annealing or mutation function in genetic algorithm). These parameters have strong impact on the quality of the

result and the time required for searching. Finding a good search parameter is in itself a complex problem and there are no *default* values to be used. If CP is used, the modeling is much more flexible than other approaches and we can benefit from the default constraint propagator and searching strategies.

The low modeling complexity of CP gives us the opportunity to have very fast prototyping and high maintainability. We could try out our new ideas and set up new experiments just by adding/deleting/modifying a few constraints or selecting another search strategy among the built ones. For example, finding an ASAP (As Soon As Possible) schedule can be achieved by searching from the lowest value, and finding an ALAP (As Late As Possible) schedule can be achieved by searching from the largest value.

2.2 Suitable for Scheduling Problem

As stated in section 1, the schedule term ($T_{max} - t_i \cdot f_i$ in 2) contributes much more than the FIMP type selection term ($2M_i$ in 2) to the number of solutions ($|D|$). Since CP is suitable for finding solutions to scheduling problems, it is suitable in our case.

2.3 Easy to be Integrated

Since CP is a programming paradigm and it is rooted in computer science, a CP solver usually has interfaces to a number of programming languages. For example, in the SYLVA project, we choose the CP solver from Google's or-tools [6]. It has interfaces to C++, JAVA, Python and C#. Another popular solver GECODE [7] has interfaces to ECLiPSe, AMPL, YAP Prolog, Python, Haskell, Ruby and Common Lisp.

In our case, most of the SYLVA components are implemented in C#. If the DSE problem modeling and solving is also implemented in C#, model translation (converting CSOP model in C#to the format the solver supports) and the resulting deserialization (converting CSOP solution to an C#object) can be eliminated.

3 Constraint Satisfaction Optimization Problem Model

In SYLVA, we have two Constraint Satisfaction Optimization Problem (CSOP) models. The first one (named as P_1) is for the SDF to HSDF conversion. The other one (named as P_2) is for the Design Space Exploration (DSE). In the rest of this section, we use the following annotations. The number of actors in the SDF graph is denoted as $|A|$ and the number of edges in the SDF graph is denoted as $|E|$. The ith SDF actor is denoted as a_i and the jth SDF edge is denoted as e_j.

3.1 P_1, SDF to HSDF Conversion

Variables in P_1 are the set of function execution counts X. The ith variable $x_i \in X$ represents the number of executions of the ith SDF actor a_i. $\forall i \quad x_i \in N_{\geq 0}$

($N_{\geq 0}$ stands for Natural number that greater or equal to zero). The number of variables $|X|$ equals to the number of SDF actors $|A|$. In our model, we constraint the possible values to be less than $2^{32}-1$. Although the number of possible values is large, the constraints will result in a small amount of branches when solving P_1, since we only need the simplest valid solution. It can be found by taking the smallest value in the domain of the variable associated with the root node, propagating this choice and iterating. For example, the conversion in Fig. 1 only has one branch.

Constraints in P_1 are such that all edges have matched data rate. Denoting the set of constraints of P_1 as C_1, the number of constraints $|C_1|$ equals to the number of SDF edges $|E|$.

Before explaining the implementation of the constraints, we need to introduce topology matrix T. T is a $|E| \times |A|$ matrix. Each row of T represents one edge and each column of T represents one SDF actor. The ith row of T is denoted as T_i and the jth element in T_i is denoted as $T_{i,j}$. $T_{i,j}$ is the number of data tokens produced or consumed by the jth SDF actor a_j on the ith edge e_i. Also denoting $a_{s,i}$ as the source actor in e_i and a_d as the destination actor in e_i, the value of $T_{i,j}$ is defined in 3.

$$T_{i,j} = \begin{cases} s_i & a_j = a_{s,i} \\ -d_i & n_j = a_{d,i} \\ 0 & n_j \neq a_{s,i} \text{ and } n_j \neq a_{d,i} \end{cases} \tag{3}$$

The SDF to HSDF conversion constraint can be expressed by 4.

$$X \cdot T_i = 0 \qquad \forall \quad 0 \leq i < |A| \tag{4}$$

For example, the topology matrix T of the SDF graph illustrated in Fig. 1(a) is listed in 5. The first row is for the edge from a to b. The second row is for the edge from d to b. The last row is for the edge from b to c. The solution for the SDF graph shown in Fig. 1(a) is $v_1 = [1, 2, 4, 4]$.

$$P = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ 0 & 2 & 1 & 0 \end{bmatrix} \tag{5}$$

HSDF Construction can be done by the pseudo-code shown in Algorithm 1. It has two steps. The first step (line 6 to 8) is to create HSDF actors based on the value of X. The second step (line 9 to 31) is to create edges based on the original SDF graph. Note that in this article, we assume that we only have multiple source actors to single destination actor or single source actor to multiple destination actors data communications. It is formulated by 6, where $x_{s,i}$ and $x_{d,i}$ are the number of executions of the source and destination actors for e_i.

$$(x_{s,i} - \lfloor x_{d,i}/x_{s,i} \rfloor \cdot x_{s,i}) \cdot (x_{d,i} - \lfloor x_{s,i}/x_{d,i} \rfloor \cdot x_{d,i}) \equiv 0 \qquad \forall \quad 0 \geq i < |E| \tag{6}$$

3.2 P_2, Design Space Exploration

In the current development stage, the number of used FIMP instances is not decided using CP but an exhaustive search. All the *load balanced* SDF schedules (which FIMP for which HSDF actors) are checked from the most parallel one (each FIMP instance executes one HSDF actor) to the most serial one (each FIMP instance executes all HSDF actors for the same SDF actor).

The *load balance* is defined as that HSDF actor executions should be distributed equally or approximately equally on proper FIMPs. It is formulated by 7, where $A_{i,f}$ is the set of HSDF actors to be executed by the fth FIMP for SDF actor a_i and F_i is the set of FIMP instances for executing all HSDF actors derived from a_i.

$$0 \le |A_{i,f}| - \lfloor x_i/|F_i| \rfloor \le 1 \qquad \forall \quad 0 \le i < |A| \text{ and } 0 \le j < |F_i| \tag{7}$$

The algorithm to generate the load balanced SDF schedules is shown in Algorithm 2 line 1 to 10. The used functions: **PossibleNumbers**, **Modify** and **Generate** are defined in line 12 to 23. Each SDF schedule produced by the **Generate** function is for further design space exploration.

Variables in P_2 are denoted as V and it has four parts, which are FIMP type selection matrix FT, buffer usage indicator vector BU, execution start time vector T_s and buffer end time vector T_{be}.

The first part FT represents the FIMP types. FT is a $M_{max} \times |A|$ matrix, where M_{max} is the maximum number of FIMPs for implementing one function. The ith row ft_i represents the FIMP type selection vector for SDF actor a_i. The possible values of the elements in FT are 0 (the FIMP is not used) or 1 (the FIMP is used). All the FIMP instances for the same SDF actor are in the same FIMP type. For the SDF graph example in Fig. 1(a), if the numbers of possible FIMPs to implement actors [a, b, c, d] are [4, 3, 2, 5]. Thus, $M_{max} = 5$.

The second part BU represents the output buffer usage. The ith element is bu_i and it represents the buffer usage for SDF actor a_i. BU is a vector with $|A|$ elements. The value of each element in BU can be 0 (one FIMP has one output buffer) or 1 (one HSDF actor has one buffer).

The third part T_s represents the execution start time for all HSDF actors. It is a vector and $|T_s = |A_H$. For the ith element $t_{s,i}$ in T_s, its value is set to integers from 0 to $T_{max} - t_{i,max}$, where $t_{i,max}$ is the maximum execution time of a_i and it is determined by the FIMP types for implementing a_i. Since other time values (t_{ie}, t_{os} and t_e in Fig. 3) are in constant relation to t_s, we do not need to search them as well.

The fourth part T_{be} represents the time to free the output buffer of each HSDF actor. It is a vector and $|T_{be} = |A_H$. The ith element in T_{be} is denoted as $t_{be,i}$ and its value can be integers from 0 to $T_{max} - 1$.

Therefore, $|V| = M_{max} \cdot |A| + |A| + |A_H| + |A_H|$. In the example shown in Fig. 1, $|V| = 5 * 4 + 4 + 10 + 10 = 44$.

Algorithm 1. HSDF Construction

```
1:  A is the actor set of the SDF graph
2:  E is the edge set of the SDF graph
3:  A_H is the actor set of the HSDF graph
4:  E_H is the edge set of the HSDF graph
5:  initialize A_H = E_H = empty array
6:  for all actor a_i in A do
7:      for j in 0 to x_i − 1 do
8:          A_H i,j = a_i
9:  for all edge e_i in E do
10:     p is the index of the source actor a_s in e_i
11:     q is the index of the destination actor a_d in e_i
12:     s_i is the number of source data token of e_i
13:     d_i is the number of destination data token of e_i
14:     if x_s > x_d then
15:         for j in 0 to x_d − 1 do
16:             for k in 0 to x_s/x_d − 1 do
17:                 add a new edge e_n to E_H
18:                 s_n is the number of source data token of e_n
19:                 d_n is the number of destination data token of e_n
20:                 a_s of e_n is A_H p,j*x_s/x_d+k]
21:                 a_d of e_n is A_H q,j
22:                 s_n = d_n = s_i
23:     else
24:         for j in 0 to x_s − 1 do
25:             for k in 0 to x_d/x_s − 1 do
26:                 add a new edge e_n to E
27:                 s_n is the number of source data token of e_n
28:                 d_n is the number of destination data token of e_n
29:                 a_s of e_n is A_H p,j
30:                 a_d of e_n is A_H q,j*x_d/x_s+k]
31:                 s_n = d_n = d_i
```

Constraints in P_2 can be sorted into six categories. They are automatically generated based on the user input.

1. The first category is data dependency for execution. A function can only start its execution after all the dependent functions are complete. For example in Fig. 4, the actor b_0 should start execution after actor a_0 is complete. This category of constraints is formulated in 8, where $t_{s,d}$ is the execution start time of the destination HSDF actor and $t_{e,s}$ is execution end time of the source HSDF actor. For the example, it is expressed as $t_{s,b_0} > t_{e,a_0}$.

$$t_{s,d} > t_{e,s} \qquad \forall \text{ edges in } E_H. \tag{8}$$

There are $|E_H|$ number of constraints in this category. $|E_H|$ is the number of edges in the HSDF graph.

Algorithm 2. HSDF Schedule

```
1: A is the actor set of the SDF graph
2: F is the FIMP instance set for the SDF graph
3: ∀ 0 ≤ i < |A|, F_i is the set of FIMP instances used for a_i
4: F_max is the set of maximum numbers of FIMP instances used for A
5: for all actor a_i in A do
6:     F_max,i = x_i
7: for all actor a_i in A do
8:     |F_i| = F_max,i
9:     for all n in PossibleNumbers(F_max,i) do
10:        Modify(i, n)
11:        Generate()
12:
13: PossibleNumbers(u):
14: output an array of numbers {n | 0 < n ≤ u and ⌊u/n⌋ * n = u}
15:
16: Modify(i, n):
17: if |F_i| > n then
18:     ratio r = F_i/n
19:     for all actor a_j in A do
20:         F_i = ⌈F_i/r⌉
21:
22: Generate():
23: for all actor a_i in actor set of the SDF graph do
24:     equally and linearly assign HSDF actors that are derived from a_i to F_i FIMPs
```

2. The second category is data dependency for output buffer. The producing and consuming actors cannot output and input at the same time to/from the output buffer. For example in Fig. 4, the actor b_0 should complete its input phase before the buffer end time of actor d_0. This category of constraints is formulated in 9, where $t_{ie,d}$ is the input end time of the destination HSDF actor and $t_{be,s}$ is output buffer end time of the source HSDF actor. For the example, it is expressed as $t_{ie,b_0} \leq t_{be,d_0}$.

$$t_{ie,d} \leq t_{be,s} \qquad \forall \text{ edges in } E_H. \tag{9}$$

There are $|E_H|$ number of constraints in this category.

3. The third category is resource dependency for execution. Only one actor can be executed on a FIMP instance at a time. For example in Fig. 4, the actor b_1 should start execution after actor b_0 is complete. This category of constraints is formulated in 10. $A_{i,j}$ is the HSDF actors that are executed on the jth FIMP instance for executing SDF actor a_i. $t_{s,p+1}$ is the execution start time for the actor a_{p+1}, which is the $p+1$th actor in $A_{i,j}$. $t_{e,p}$ is the execution end time for actor a_p, which is the pth actor in $A_{i,j}$. For the example in Fig. 4, $A_{d,1}$ has d_2 and d_3, and both of them are executed on FIMP instance D_1.

$$t_{s,p+1} > t_{e,p} \qquad \forall \ a_p \in A_{i,j}. \tag{10}$$

There are $|A_H|-|F|$ number of constraints in this category. $|A_H|$ is the number of HSDF actors and $|F| = \sum F_i$ $(0 \leq i < |A|)$ is the total number of used FIMP instances and it is determined in the HSDF scheduling step.

4. The fourth category is resource dependency for output buffer. One output buffer can only be written by one HSDF actor at a time. For example in Fig. 4(a), the actor b_1 cannot start writing data into the output buffer before it is freed by c_0 and c_2. This category of constraints is formulated in 11. $t_{os,p+1}$ is the output start time for the actor a_{p+1}, which is the $p+1$th actor in $A_{i,j}$. $t_{be,p}$ is the output buffer end time for actor a_p, which is the pth actor in $A_{i,j}$. $A_{i,j}$ is the HSDF actor set that is derived from the SDF actor a_i and executed on the jth FIMP instances a_i. For the example in Fig. 4(a), t_{os,b_1} should be larger than t_{be,b_0} since no extra buffer is used ($bu_b = 0$). While in Fig. 4(b), t_{os,b_1} can be smaller than t_{be,b_0} since extra buffer is used ($bu_b = 1$). There are $|A_H| - |F|$ number of constraints in this category.

$$(t_{os,p+1} > t_{be,p} \text{ or } bu_i) = True \qquad \forall\ a_p \in A_{i,j}. \tag{11}$$

5. The fifth category is cost constraints. There are area, energy and timing cost constraints.

The total *area* usage $AREA$ can be computed by 12, where $|F_i|$ is the number of FIMP instances for executing SDF actor a_i, $area_i$ is a vector that represents the area usage for all the FIMPs for executing SDF actor a_i and ft_i is the ith row of FT (FIMP type variables for a_i). The term $area_i \cdot ft_i$ is the area usage for the used FIMP instance for a_i.

$$AREA = \sum_{i=0}^{|A|-1} (|F_i| \cdot area_i \cdot ft_i). \tag{12}$$

The total *energy* cost $ENERGY$ can be computed by 13, where $energy_i$ is a vector that represents the energy cost for all the FIMPs for executing SDF actor a_i. The term $energy_i \cdot ft_i$ is the energy cost for one execution of a_i.

$$ENERGY = \sum_{i=0}^{|A|} (x_i \cdot energy_i \cdot ft_i). \tag{13}$$

The *timing* costs consist of system latency $LATENCY$ (the time between the first input data token is consumed till the last output data token is produced) and system sample interval $INTERVAL$ (the time for one system iteration). They can be computed by 14 and 15, respectively. A_H is the actor set of the HSDF graph. a_q and a_0 are the last and the first HSDF actor, respectively, in $A_{i,j}$, which is a actor set containing all HSDF actors that are derived from SDF actor a_i and executed on the jth FIMP instance for a_i.

$$LATENCY = max(t_{e,p}) \qquad \forall\ a_p \in A_H \tag{14}$$

$$INTERVAL = max(t_{be,q} - t_{os,0}, t_{e,q} - t_{s,0}) \qquad \forall\ a_q, a_0 \in A_{i,j} \tag{15}$$

In the example shown in Fig. 4(a), $LATENCY = 22$ and $INTERVAL = 14$.

$$\begin{aligned} INTERVAL &= max(t_{e,a_0} - t_{s,a_0}, t_{be,a_0} - t_{os,a_0}, ...) \\ &= max(6, 7-4, 13-2, 14-4, 17-6, 21-7) \\ &= max(6, 3, 11, 10, 11, 14) = 14 \end{aligned}$$

The area, energy and timing cost constraints are shown in 16.

$$\begin{aligned} &AREA &\leq A_{max}, \\ &ENERGY &\leq E_{max}, \\ &LATENCY &\leq T_{max}, \\ &INTERVAL &\leq R_{max}. \end{aligned} \tag{16}$$

In SYLVA, providing values to A_{max}, E_{max}, T_{max} and R_{max} is optional. If any of them is not provided by user, the default value $2^{64} - 1$ will be used. There are 4 constraints in this category as shown in 16.

6. The sixth category is the FIMP type constraint. Only one FIMP can be used to implement an SDF actor a_i. This constraint is shown in 17, where $ft_{i,j}$ is usage of the jth FIMP for implementing a_i

$$\sum_{j=0}^{M_{max}} ft_{i,j} = 1 \qquad \forall\ 0 \leq i < |A| \tag{17}$$

There are $|A|$ number of constraints in this category.

The number of constraints in P_2 is $|C_2| = 2 \cdot |E_H| + 2 \cdot (|A_H| - |F|) + 4 + |A|$. In the example shown in Fig. 1, $|C_2| = 2 * 10 + 2 * 6 + 4 + 4 = 40$.

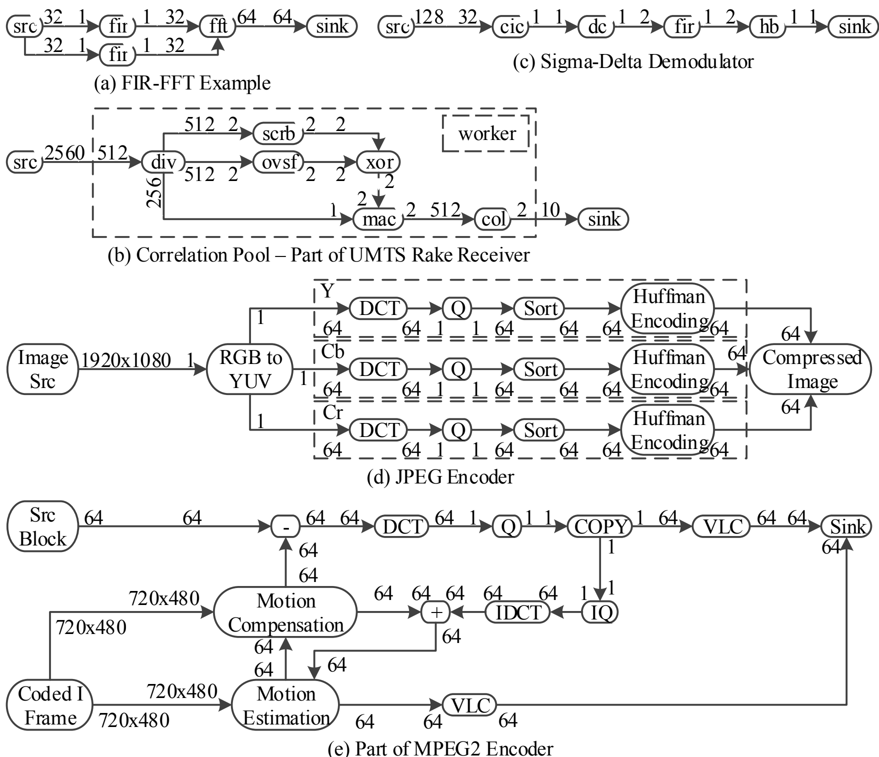
Optimization Objective is to minimize the total cost C, which is calculated by 18. K_A, K_E, K_T and K_R are four predefined constants for optimizing the solution in terms of area usage, energy cost, system latency and system sample interval, respectively. They can be provided by user or be the default value ([0, 1, 0, 0] for [K_A, K_E, K_T, K_R]).

$$C = K_A \cdot AREA + K_E \cdot ENERGY + K_T \cdot LATENCY + K_R \cdot INTERVAL \tag{18}$$

3.3 Solving CSOPs

In SYLVA, P_1 and P_2 are modeled and solved in C# by using the CP solver in or-tools from Google. For branching strategy, we set the solver to always select the first unbound variable (INT_VAR_DEFAULT) and assign the minimum possible value first (ASSIGN_MIN_VALUE).

The efficiency and efficacy of SYLVA are evaluated by synthesizing five examples (Fig. 5): 1) a sub-system composed of two FIR filters feeding an FFT, 2) a correlation pool (part of UMTS rake receiver), 3) a sigma delta demodulator, 4) a JPEG Encoder for 1920 × 1080, and 5) a simplified MPEG2 Encoder for a 720 × 480 @25 frames per second.



(a) FIR-FFT Example
(c) Sigma-Delta Demodulator
(b) Correlation Pool – Part of UMTS Rake Receiver
(d) JPEG Encoder
(e) Part of MPEG2 Encoder

Fig. 5. Examples for Experiments

The individual components of the SYLVA flow have been implemented in C# and integrated at script level. This script was invoked for each of the five examples (in the form of SDF graphs) with the maximum sampling interval (R_{max}) and the maximum total latency (T_{max}) as command line parameters. The experimental result shows that on average, the SYLVA runtime for the five examples are 15s, 13s, 20s, 74s, and 97s, respectively, and for all the examples, only one branch is required to find the SDF repetition vectors. They are much faster than the commercial high level synthesis tool we compared with. SYLVA gets speed advantage compared to commercial synthesis tools because of its use of very large grain design objects (FIMPs) that are 2-3 orders smaller than the objects used by commercial tools. This dramatically reduces the design space for SYLVA. The details of the quality of result comparison can be found in [2].

4 Conclusion and Future Work

4.1 Development Cost

Learning Cost: Before coding SYLVA, one of the authors has taken a CP course, which lasts for two months. The author also spent an additional month

to get familiar with the C# port of the CP solver in Google's or-tools. **Time Cost:** By using CP, the first SYLVA release came out after one month of intensive coding in C#. The time spent on the Design Space Exploration (DSE) CSOP model is only one week and it includes coding, debugging and experimenting. By using CP, modifying and maintaining the DSE model are quite straightforward and simple. We only need to add/remove/modify the concerned constraints. **Software Cost:** The used software copies are all free. The text editor and or-tools from Google are free software and the Windows SDK is also free to use.

4.2 Usage Difficulty

Using SYLVA to synthesize a system SDF graph into a hardware description is quit simple and does not require any knowledge of CP. The user is required to provide an SDF graph. Providing A_{max}, E_{max}, T_{max}, R_{max}, K_A, K_E, K_T and K_R are all optional. In most of the cases of hardware design, the final system implementations should be optimized to have minimal area or energy. In this case, K_A and K_E should be 1, while K_T and K_R are 0's.

4.3 Conclusion

By using CP, we saved a lot of time on implementing the model of the problem. Compared to other approaches, e.g. integer linear programming or evolutionary algorithms, CP provides a straightforward framework for modeling and solving problems. After three months of non-intensive learning of CP fundamentals and a CP solver, we were able to write a complex CSOP model, which has four categories of variables and six categories of constraints in a short time. We hope the work in this article could serve as a good application example of CP.

4.4 Future Work

Currently, we are updating SYLVA to support more features, such as scenario-aware SDF that supports dynamic streaming and signal processing applications, and I/O data structure matching that match not only the number of data tokens but the data structure of the data tokens to improve the timing performance of the system.

References

1. Lee, E., Messerschmitt, D.: Synchronous Data Flow. Proceedings of the IEEE 75(9) (September 1987)
2. Li, S., Farahini, N., Hemani, A., Rosvall, K., Sander, I.: System Level Synthesis of Hardware for DSP Applications Using Pre-Characterized Function Implementations. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 1–10 (September 2013)

3. Bonfietti, A., Benini, L., Lombardi, M., Milano, M.: An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 897–902 (March 2010)
4. Li, S., Hemani, A.: Global Interconnect and Control Synthesis in System Level Architectural Synthesis Framework. In: Euromicro Conference on Digital System Design (DSD), pp. 11–17 (September 2013)
5. Li, S., Malik, J., Liu, S., Hemani, A.: A Code Generation Method for System-Level Synthesis on ASIC, FPGA and Manycore CGRA. In: Proceedings of the First International Workshop on Many-core Embedded Systems (2013)
6. Operations Research Tools from Google, https://code.google.com/p/or-tools/
7. Gecode: generic constraint development environment, http://www.gecode.org/