DYNAMIC CONFIGURATION OF DATAFLOW GRAPH TOPOLOGY FOR DSP SYSTEM DESIGN

Dong-Ik Ko and Shuvra S. Bhattacharyya

{dik,ssb}@eng.umd.edu

Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, 20742, USA.

ABSTRACT

Dataflow is widely used for designing DSP applications. Despite its intrinsic advantages, one weak point is its difficulty in flexible expression of applications with data dependent change in execution structure. This paper suggests an approach to providing dynamically configured dataflow graph topologies using a new modeling and synthesis technique called DGT (Dynamic Graph Topology). DGT builds on PSDF semantics [1]. All possible graph topologies for a given graph are obtained at a compile time and the corresponding graph based on parameters and data is dynamically set up in an efficient manner at runtime before the invocation of the associated graph. Systematic methods for reducing code and buffer size are applied based on characteristics of each configured graph. We have compared DGT against conventional modeling approaches through a detailed case study of an MPEG 2 video encoder system, and our experiments demonstrate the efficiency of the DGT approach.

1. RELATED WORK

To handle data driven changes in execution structure, several dataflow models such as CDDF [11], BDF [4], and BDDF [9], have been proposed. CDDF uses control tokens to determine the token transfer at an actor port. However, determination by a control token is applied to the actor in the next phase of execution, therefore, control tokens are not present at the moment that the actual phase is determined. BDDF introduces dynamic ports and an upper bound is provided for the data rate so that each dynamic port can keep the model bounded. However, control flow depends on FSMs. Using FSMs for minor changes of control flow with dataflow graphs can make application models unnecessarily complicated and result in limited flexibility. BDF provides "SWITCH" and "SELECT" actors to determine control flow. For satisfying bounded memory and consistency, a symbolic function of probability is introduced. This function increases the complexity of solving the balance equations (for verifying sample rate consistency), and results in the possibility of "weak consistency," which is less desirable in an implementation.

To provide for more powerful and efficient data dependent execution related to application mode changes, where entire graphs or subsystem are replaced or reconfigured at run time, this paper tackles dynamic set-up of dataflow graph topologies before the graphs are invoked. All configurations of possible graph topologies are pre-computed at compile time and stored for usage at run time. At runtime, the initialization step of DGT generates an appropriate graph topology based on parameters extracted from data being delivered and picks up a pre-computed schedule to fit the current parameter configuration. However, not all configurations are valid or can be obtained at a compile time. Some configurations may cause deadlock or inconsistency or may not be predictable at compile time. Reconfiguration of dataflow graphs is carefully considered in [10]. [10] analyzes the reconfiguration of a model based on behavioral types and extracts the *least change context* to check approximate semantic constraints. This paper statically checks the validity of each configuration like [10] and keeps the scheduling results for use at run time. The main distinguishing feature of DGT is that it efficiently supports multi-function applications by configuring graph topologies dynamically. There are two kinds of multi-function applications. The first, which we call type-I applications, are exclusive-or applications, where only one graph topology is selected from multiple sets of possible graph topologies for a given application. The other, which we call type-II applications, are concurrent applications where two or more applications with different graph topologies are running at the same time. This paper focuses on type-I (exclusive-or) application for experimentation of DGT. For synthesis of type-I applications, [5] extracted *commonality measures* of each actor and used these values to determine a hardware bias of each actor by hardware oriented partitioning. This paper focuses on software implementation, and applies novel scheduling techniques based on graph characteristics to reduce code and buffer size, which is critical for DSP software. The DGT approach provides efficiency and flexibility in modeling applications with data driven change of graph topology from runtime parameter changes by using pre-computed information (information related to graph topology, scheduling, code/buffer size, bounded memory, etc.).

2. DYNAMIC GRAPH TOPOLOGY

2.1 Brief description of PSDF graphs

PSDF specification Φ consists of three distinct graphs: 1) the *init* graph Φi; 2) the *subinit* graph Φs; and 3) the *body* graph Φb. Intuitively, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parameters. The init graph is invoked prior to each invocation of the associated (hierarchical) parent subsystem, `parent(Φ)`, while the subinit graph is invoked prior to each invocation of the associated body subsystem Φb, thus allowing for two distinct "frequency levels" of reconfiguration control.

2.2 DGT (Dynamic Graph Topology) specifications

As applications for embedded systems grow more complicated, the requirement of dynamic on/off of actors and ports of actors as well as the change of transfer rates(production and consumption rates) on dataflow edges is unavoidable. To support dynamic change of graph topologies, actors, ports of actors and transfer rates should be considered to be adaptable based on the delivered data. Dynamic change of a graph topology requires run-time scheduling, which potentially causes problems of execution time overhead. To alleviate this overhead, this paper provides for dynamic change of graph topologies through schedules that are pre-computed at a compile time. DGT is based on PSDF semantics [1],[6], but is significantly more flexible than PSDF in that it allows graph actors and edges to be treated as dynamic parameters as well as the more standard types of parameters supported in the dynamic reconfiguration capabilities of PSDF. Therefore, in DGT, the transfer rate of each port of a graph, itself, is determined by a

special subgraph, called the *init* graph, as in PSDF [1], so that the consumption rate and production rate of each port of the graph can be determined before the invocation of the associated DGT graph. However, in DGT, the *subinit* graph Φs controls the behavior of the associated body graph by determining the graph topology of the associated body graph before the invocation of the body graph. The number of possible graph topologies is predicted at a compile time.

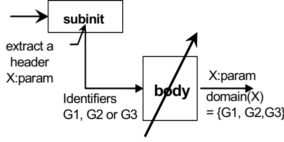

Figure 1. DGT(Dynamic Graph Topology)

Figure 2. An example of a graph under DGT

Figure 1 shows that how a subinit graph can extract appropriate header information and set up parameters (X:param) with the required information for the associated body graph. An appropriate graph is selected from a set of possible graphs($\{G_1, G_2, G_3\}$ by the subinit graph with (X:param). This mechanism is effective because many data tokens for modern DSP applications are delivered as frames with a header part and a payload part.

Here, we classify actors and ports into two categories based on the presence or absence of data driven change of their behaviors. Actors and ports that are not changed in a graph topology are called fixed actors (a_f) and fixed ports (p_f), respectively, while actors and ports having potential dynamic changes are named as varying actors (a_v) and varying ports (p_v). Here, one point that requires careful consideration is that a fixed actor(a_f) can have a varying port (p_v) since a fixed actor (a_f) can appear with different types of ports. The *subinit* graph Φs dynamically sets up varying actors and varying ports based on data being delivered and produces an appropriate graph topology for the associated body graph. Consistency and bounded memory for each possible set of graph topologies are verified at compile time. At runtime, the *subinit* graph Φs sets up an appropriate graph topology for the associated body graph and picks up an appropriate pre-computed schedule that also contains code and buffer size minimized for the configured graph. Code and buffer size minimization is obtained by a scheduling technique appropriately chosen depending on graph characteristics. In DGT, verification of validity of schedules can be performed at a compile time and valid schedules can be guaranteed and can be ready to be used at runtime without the overhead of dynamic scheduling. At runtime, *the subinit* graph Φs looks up pre-computed schedules in a table with the appropriate parameter values.

Figure 2 shows an example of how DGT is applied to configure a body graph. Here, $\{p_v^o(i, a_k)\}$ represents all the possible sets of ports to which the i_{th} varying output port of the actor a_k can be connected. $\{p_v^{in}(i, a_k)\}$ represents a counterpart of an input port. In figure 2, dotted line represents varying edges while solid lines represents fixed edges. Also, a dash filled actor represents a varying actor while a white blank actor represents a fixed actor. Each actor can have varying ports and fixed ports together. The transfer rates or connections of varying edges are data dependent while the transfer rates and connections of fixed edges are fixed. Varying edges and varying actors can be turned on or off based on the data tokens delivered.

The following equation represents a general case where the i_{th} varying output or input port of the actor a_k connects to the j_{th} input or output port of another actor a_n or does not connect to anything.

$$\{p_v^o(i, a_k)\} = \{p_v^{in}(j, a_n), \perp\}, (\{p_v^{in}(i, a_k)\} = \{p_v^o(j, a_n), \perp\})$$

This is an example of the $1st$ input port of a_6 in Figure 2.

$$p_v^{in}(1, a_6)\} = \{\{p_v^o(1, a_4), p_v^o(1, a_5)\}, p_v^o(1, a_3), p_v^o(1, a_2)\}$$

Here, $\perp$ means there are no edges from or to the associated port. The graph G ($G = G_f \cup G_v$) is made up of G_v (a graph with varying graph components) and G_f (a graph with fixed graph components). By separating from G_f parts that are common across different subsystems, possible overlapping of resources among different subgraphs can be removed.

2.3 Scheduling of DGT specifications

A DGT subsystem produces various sets of configurations for the associated body graph Φb. For each graph generated, checking of both synchrony (synchronous dataflow [8] behavior) for the duration of the configuration and bounded memory is performed. For this purpose, a graph is considered as a general fixed graph after the subinit graph configures the graph topology. All of the major configurations for the corresponding graph are captured at the compilation stage and are kept for use at runtime. The *subinit* graph Φs extracts parameters from the header part of data being processed and then sets appropriately the associated body graph Φb. For many applications, such as those involving a few to several or even dozens of different modes, the number of combinations of DGT configurations is manageable for reasonable implementation platforms. Here, the transfer rate of every port of each actor within a body graph under DGT can be changed by the associated graph Φs.

A useful restriction in the use of DGT is that when a DGT graph is embedded within a dataflow model other than DGT or PSDF, the transfer rates of interface ports of a DGT graph must generally be fixed even though the graph topology inside the DGT subsystem can be vary dynamically. This assumption allows DGT graphs to be embedded easily in other dataflow models with the external appearance of simple SDF actors. Therefore, the transfer rates of input/output ports of the DGT graph, itself, should be set by the *init* graph Φi before the DGT graph is invoked and should be kept invariant during the entire iteration of the graph.

Figure 3 shows an example that illustrates DGT scheduling within SDF. The DGT graph Φ takes two tokens and produces two tokens. Therefore, the schedule for Figure 3 will be like $2s \cdot \Phi \cdot 2e$. However, by looking into the DGT graph Φ, we see that the actor b is a varying actor that can be removed by the subinit graph Φs on demand. Also, the transfer rates of actor b are not fixed. The actor a has one output port, which is a varying port. Therefore, the actor a can be connected to either the actor b or the actor c. The actor c has one varying input port and one fixed output port. The actor c consumes one token either from actor a or actor b and produces two tokens to a fixed output port. The

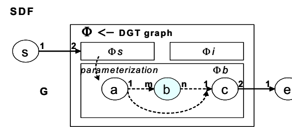

Figure 3. DGT graph under SDF

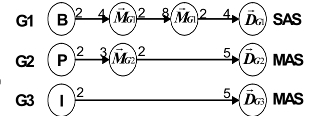

Figure 4. Part of an MPEG2 video encoder

schedule of the DGT graph Φ can be either $ma \cdot b \cdot nc$ or $a \cdot c$. The schedule for the graph G is $2s \cdot \Phi \cdot 2e$ and the schedule for the graph Φ is either $ma \cdot b \cdot nc$ or $a \cdot c$. The schedule for each graph is hierarchically maintained in this manner. Here, the two

schedules for the graph Φ are SAS (Single Appearance Schedule)[3] where each actor appears only once. The following section shows how different scheduling techniques are applied systematically based on characteristics of the configured graphs.

2.4 Minimization of code and buffer requirements

According to graph characteristics and the granularity (complexity) of each actor, efficient scheduling considering both code size and buffer memory requirements is important when synthesizing implementations. Since a DGT system supports runtime adjustment of pre-computed schedules, decisions on the methods for minimizing code and buffer requirements can be made statically. For an application graph, the ratio of code size vs buffer size as well as graph characteristics are important factors to select an appropriate technique for efficient minimization of both code and buffer size. For example, for an application with a very small code size but requiring high buffer size, minimizing code size by SAS (Single Appearance Schedule) is not likely to lead to a cost-effective solution. Instead, a carefully-constructed MAS (Multiple Appearance Schedule) is likely to be a better choice due to the advantage of further buffer size reduction at the expense of some code size increase. In our DGT synthesis approach, for efficient multiple appearance schedule generation, we have adapted the MAS approach of [7], and for SAS generation, techniques from [1], [2] and [3] are applied. For selection between MAS and SAS implementation, we have formulated a normalized criterion (SS:Schedule Selector) to determine the most appropriate technique.

$$SS = \gamma_\mu \times \mu + \gamma_\tau \times \tau$$

μ is the uniformity metric of [7] (explained below) and τ is the ratio of total code size to the average data frame size obtained based on simulation. γ_μ and γ_τ are user-defined weight values and are chosen based on simulation. μ is proportional to the number of edges whose transfer rates are multiples of one another. A high value of μ reflects potentially low opportunity for buffer size reduction using the techniques of [7]. τ suggests which factor between code size and buffer size is more important to reduce the overall memory requirements. A graph with a higher τ suggests that a scheduling technique that is more efficient in reducing code size produces a better result rather than a buffer-oriented technique. Consequentially, a high SS value suggests that an SAS is appropriate for the graph.

Figure 4 shows part of an MPEG2 encoder modeled using our DGT technique. Some of the actors can operate with different parameters and transfer data at rates depending on the graph(G) in which the actor is included. Those actors are symbolized as $\vec{X}_G$. In Figure 4, $\vec{M}$ represents MC (motion compensators) and $\vec{D}_G$ represents a DCT (Discrete Cosine Transform). In MPEG2, the B frame requires two MCs and the P frame requires one MC, while the I frame does not need a MC. Therefore, three different graph topologies are required within the application, and the particular topology to use at a given time depends on the picture frame type (I, P, or B).

Each graph topology has different SS values depending on the characteristics the graph. For G1 of B frame, SAS implementation is selected, while for G2 of P frame and G3 of I frame, MAS implementation is selected. In Figure 4, the behaviors of the actor $\vec{M}_G$ and the actor $\vec{D}_G$ can be changed depending on the graph characteristics and the change of parameters, while other actors are invariant.

From a DGT representation, we can often reduce code size by removing overlapping graph components across graph sets. If m is the number of common actors in graphs with different configurations, and λ_i is the number of graphs (G_i) including the i_{th} common actor (C_i).

$$ReducedCodeSize = \sum_{i=1}^{m} \sum_{j=1}^{\lambda_i - 1} codeSize(C_i)$$

2.5 Operational semantics of DGT

Figure 5 shows the operational semantics of DGT operating with any type of dataflow model. Because of its ability to operate with different types of dataflow models, DGT is more accurately characterized as a meta-modeling technique. Each hierarchical actor (Φ) in a DGT system also can be viewed as an independent graph and can have its own schedule. In our implementation of DGT, we maintain schedules in a hierarchical manner. Therefore a graph (G) has the schedule for itself and also maintains schedules for each hierarchical actor(Φ) under the graph (G). Each hierarchical actor Φ under G also maintains the schedule for itself and schedules for graphs representing every hierarchical actor Φ_{sub} inside Φ. This way, the schedule for the graph G and schedules for sub graphs of Φ s inside G are maintained in a hierarchical way until graphs in the lowest level of the hierarchy are scheduled.

The function $scheduleX(G)$ is a function to schedule a graph G. For all general hierarchical actors (Φ) inside G except Φ s of DGT, $scheduleX$ is applied. The function $scheduleDGT$ is applied for Φ of DGT within G. Then $linkSC$ is applied to have the schedule for the graph G, itself and schedules for Φ s in G kept linked together. The function $setGraphTopology$ in $scheduleDGT$ generates the corresponding graph with given parameters. Ultimately, $schedulerXDF$ in a $scheduleX$ generates an appropriate schedule based on the graph topology along with code and buffer size suitable for each graph. For each configured graph, type checking of the given graph is performed and then if SS is bigger than $Threshold_{SS}$ for selecting an scheduling technique, the chosen SAS based technique ($SASTechnique$) is applied. Otherwise, the chosen MAS based technique ($MASTechnique$) is chosen.

3. EXPERIMENTAL RESULTS

In our experiments, we developed an MPEG2 video encoder as an application example. An MPEG2 video encoder has some different operational blocks depending on the picture frame, but shares most of the blocks across picture frames (I, B or P frame). We compared the total memory usage of a DGT graph implementation with a conventional separate-graph approach. A separate graph approach uses a combination of SDF and FSM. Each SDF graph processes a different picture frame. The DGT method selects different scheduling methods (SAS or MAS) depending on graph characteristics. For obtaining the code size, we used the Texas Instruments Code Composer simulator of the 64XX series processor. As the frame size increases, the impact of buffer size on total memory usage becomes larger than the impact of code size. We applied SAS, MAS and a combination of SAS and MAS to each case. In C3 and C6, (see Table 1) while SAS is selected for both 128*128 and 256*256, either SAS or MAS is selected for each picture frame (I, B and P) dynamically for a frame larger than 256*256. This is because a trade-off between code size and buffer size exists in the vicinity of 480*720 size.

Table 1 shows that the DGT approach reduces total memory usage from 60% to 67% compared with a separate graph approach through shared code and the streamlining of scheduling methods to

```
function scheduleX(G) {
   for each Φ_i of DGT in G
      sched_Φi = scheduleX(Φ_i[i])
   end for
   shed_G = scheduleXDF(G)
   for each Φ in G
      if(Φ under DGT)
         sched_Φ[i] = scheduleDGT(Φ[i])
      else
         sched_Φ[i] = scheduleX(Φ[i])
   end for
   sched_G = linkSC(linkSC(shed_G, sched_Φ), sched_Φi)
   return sched_G
}
function schedulerXDF(G) {
   if(SS > Threshold_SS)
      sched_G = SASTechnique(G)
   else
      sched_G = MASTechnique(G)
   return sched_G
}
function scheduleDGT(Φ) {
   sched_Φs = scheduleX(Φ_s)
   setUpPortTransferRate(Φ) //in & out port of Φ
   C_b = getParamConfigSets(Φ_b)
   for each C_b in Φ_b
      Φ*_b = setGraphTopology(C_b, Φ_b)
      sched_Φ*b[i] = scheduleX(Φ*_b)
   end for
   sched_Φ = linkSC(sched_Φ*b, sched_Φs)
   return  sched_Φ
}
function setGraphTopology(Φ, C) {
    determineG_vTopology(Φ, C)
   for each a in Φ
      for each p^o_v in a
       {e_v} = connectEdge(p^o_v(i, a_v))
      end for
      for each p^in_v in a
         {e_v}= {e_v} ∪ connectEdge(p^in_v(i, a_v))
      end for
   end for
   Φ_v = {a_v} ∪ {e_v}
   Φ = Φ_v ∪ Φ_f //Φ_v , Φ_f each varying and fixed graph.
   return Φ
}
```

. $Threshold_{SS}$ is obtained based on simulation

Figure 5. Operational semantics of DGT operating with any type of dataflow model

fit graph characteristics. The runtime overhead for finding a proper schedule for each graph topology is only $\Omega(N) + \Omega(m)$, where m is the number of varying graph components (varying actors and varying edges) and N is the number of possible schedules for each DGT graph depending on the topology, which is relatively modest compared with the complexity of typical signal/image processing actors.

4. CONCLUSIONS AND FUTURE WORK

This paper develops efficient support for dynamic graph topologies for dataflow graphs requiring different execution structures based on dynamic parameters, and data being processed. In addition to providing efficient and flexible support for multiple modes of system operation, DGT allows us to reduce overall memory size by systematically sharing code and applying tailored scheduling methods across the different graph topologies that make up a DGT application. Useful directions for future work include integrating DGT with other dataflow models as a meta-modeling technique, and implementation of concurrent applications through DGT semantics under resource and performance constraints.

Table 1. Memory usage comparison

Frame Size		DG			SG		
		C1	C2	C3	C4	C5	C6
128 * 128	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	1,557	1,429	1,557	4,667	4,283	4,667
	Total	28,026	33,375	**28,026**	68,008	84,056	**68,008**
256 * 256	Code	26,469	31,946	26,469	63,341	79,773	63,341
	Buffer	6,173	5,661	6,173	18,515	16,979	18,515
	Total	32,642	37,607	**32,642**	81,856	96,752	**81,856**
480 * 720	Code	26,469	44,903	31,393	63,341	118,645	94,180
	Buffer	52,852	19,991	21,788	158,551	59,967	65,364
	Total	79,321	64,894	**53,181**	221,892	178,612	**159,544**
768 * 1024	Code	26,469	58,074	44,564	63,341	158,157	133,692
	Buffer	130,680	45,320	49,397	392,035	135,955	148,192
	Total	157,149	103,394	**93,961**	455,376	294,112	**281,884**
1080 * 1920	Code	26,469	58,074	50,041	63,341	158,157	150,124
	Buffer	1,817,064	100,940	100,937	5,451,187	302,815	302,524
	Total	1,843,533	159,014	**150,978**	5,514,528	460,972	**452,648**

. DG: DGT approach, SG: Separate graph approach(FSM+SDF), C1: SAS, C2: MAS,
. C3: SAS+MAS, C4: SAS, C5: MAS, C6: SAS+MAS

5. REFERENCES

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing, 49(10):2408-2421, October 2001.

[2] S. S. Bhattacharyya, R. Leupers, P. Marwedel, “Software Synthesis and Code Generation for Signal Processing Systems”, Tech. Report UMIACS-TR-99-57, Institute for Advanced Computer Studies, University of Maryland, College Park, September, 1999.

[3] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[4] J. T. Buck, E. A. Lee, “Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model”, *Proc. ICASSP*, April, 1993.

[5] A. Kalavade and P. A. Suhrahmanyam, "Hardware / Software Partitioning for Multi-function Systems", *Proc. International Conference on Computer Aided Design*, pp. 516-521, Nov. 1997.

[6] D. Ko and S. S. Bhattacharyya. Modeling of block-based DSP systems. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 381-386, Seoul, Korea, August 2003.

[7] M. Ko, P. K. Murthy, and S. S. Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Processors*, Amsterdam, The Netherlands, September 2004.

[8] E.A. Lee, D.G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing”, *IEEE Transactions on Computers*, February, 1987.

[9] M. Pankert, O. Mauss, S. Ritz, H. Meyr, “Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis,” Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, pp 449-452, Adelaide, Australia, April 19-22, 1994.

[10] Stephen Neuendorffer and Edward Lee, “Hierarchical Reconfiguration of Dataflow Models”, Conference on Formal Methods and Models for Codesign (MEMOCODE), June 22-25, 2004.

[11] P. Wauters, M. Engels, R. Lauwereins, J.A. Peperstraete, “Cyclo-dynamic dataflow,”4th EUROMICRO Workshop on Parallel and Distributed Processing, Braga,Portugal, January, 1996.