In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, 2012, to appear.

Partial Expansion Graphs: Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications

George F. Zaki, William Plishker, Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering
University of Maryland College Park
College Park, Maryland
{*gzaki, plishker, ssb*}*@umd.edu*

Frank Fruth
Multicore and Communication Infrastructure Business Unit
Texas Instruments Incorporated
Germantown, Maryland
ffruth@ti.com

***Abstract*—Emerging Digital Signal Processing (DSP) algorithms and wireless communications protocols require dynamic adaptation and online reconfiguration for the implemented systems at runtime. In this paper, we introduce the concept of Partial Expansion Graphs (PEGs) as an implementation model and associated class of scheduling strategies. PEGs are designed to help realize DSP systems in terms of forms and granularities of parallelism that are well matched to the given applications and targeted platforms. PEGs also facilitate derivation of both static and dynamic scheduling techniques, depending on the amount of variability in task execution times and other operating conditions. We show how to implement efficient PEG-based scheduling methods using real time operating systems, and to re-use pre-optimized libraries of DSP components within such implementations. Empirical results show that the PEG strategy can 1) achieve significant speedups on a state of the art multicore signal processor platform for static dataflow applications with predictable execution times, and 2) exceed classical scheduling speedups for applications having execution times that can vary dynamically. This ability to handle variable execution times is especially useful as DSP applications and platforms increase in complexity and adaptive behavior, thereby reducing execution time predictability.**

***Keywords*-Dataflow Graphs, Digital Signal Processing, Multiprocessor Dynamic Scheduling.**

I. Introduction

Design methods for digital signal processing applications, such as Long Term Evolution (LTE), codecs for video and audio players, and Network Intrusion Detection systems (NIDs), have been evolving continuously over the last decades. Multiple platforms can be selected to implement such systems. Factors such as time to market (i.e., ease of development), power consumption, system throughput, latency, and other metrics guide the selection of the final processing unit type. Classically, designers have been able to achieve such objectives using single core processors. Following Moore's law and the necessity to limit the dissipated power on a single ship, performance gains in these processing platforms have been coming in recent years from increasing the number of cores on a single die instead of increasing the frequency of a single core.

Following this historical evolution of programmable platforms, developers are required to migrate to multicore platforms their libraries of kernels that were optimized originally to target single core platforms. For such migration to multicore solutions, efficient *scheduling* of the algorithm kernels to the available computing units is required. By scheduling, we mean the assignment of kernels to cores, and the ordering of kernels that share the same core. Scheduling is known to be NP complete except for certain restrictive special cases, and therefore, in practice, a combination of heuristics, designer experience, and high (non-polynomial) complexity algorithms are employed to derive scheduling solutions.

Exploring different levels of parallelism in DSP applications can be facilitated by modeling the applications using dataflow graphs (e.g., see [2]). Various forms of DSP-oriented dataflow models exists, where Synchronous Dataflow (SDF) is among the most commonly used [11]. In SDF, the number of tokens (data values) produced and consumed at each actor (dataflow graph vertex) port is constant across each firing (invocation) of the associated actor.

Multiprocessor scheduling for an SDF graph conventionally involves a transformation of the graph to its equivalent Homogeneous SDF (HSDF) form. Such *HSDF expansion* is performed to more fully expose the parallelism in an SDF representation, including task and data parallelism, as explained in Section III. However, the HSDF expansion transformation may in general produce an exponential increase in dataflow graph size — the numbers of actors and edges (graph size) in the HSDF equivalent graph is in general not polynomially bounded in the size of the corresponding SDF graph. This expansion can thus lead to very slow or memory consuming scheduler performance, which limits the effectiveness of design space exploration and other analysis tasks. Furthermore, considering the typical number of cores in contemporary DSP platforms, the full parallelism exposed by an HSDF expansion may not be beneficial because such parallelism (e.g., tens to hundreds of parallelizable firings per scheduling iteration) may over-

whelm the number of available cores (e.g., the latest Texas Instruments TMS320C6678 processor has eight cores).

In this paper, we present a scalable dataflow graph intermediate representation, called *partial expansion graphs* (*PEGs*), and we present an implementation and scheduling strategy that utilizes PEGs to realize DSP systems efficiently on multicore platforms. Intuitively, PEGs provide benefits during the scheduling process by allowing designers and design tools to tune the trade-offs involving dataflow graph (intermediate representation) complexity and exposed parallelism based on relevant considerations, such as the amount of parallelism that is available in the target platform. This allows scheduler construction and exploration to proceed more effectively compared to always operating on fully expanded HSDF graphs (high complexity / high parallelism exposure) or unexpanded SDF graphs (low complexity / low exposure), which can be viewed as the extremes in the space of possible PEG representations. We develop methods to tune PEG representations based on application and architecture characteristics, systematically integrate PEG-based scheduling into real time DSP operating systems, naturally express diverse forms of parallelism in dataflow graphs, and perform buffer management operations associated with PEG representations to dynamically distribute time-varying firing loads (i.e., when execution times can vary dynamically).

II. Background

Dataflow models of computation, such as synchronous dataflow (SDF) [11] and cyclo-static dataflow [3], are widely used in design, analysis and implementation of DSP systems. A dataflow model of a DSP application is a useful way to capture important data dependency relationships and to express parallelism between system kernels.

A dataflow graph G consists of a set of vertices V and a set of edges E. The vertices or *actors* represent computational kernels, and the edges represent FIFO buffers that can hold data values, which are encapsulated as *tokens*. Depending on the application and the required level of model-based decomposition, actors may represent simple arithmetic operations, such as multipliers or more complex operations, such as turbo decoders.

A directed edge $e = (v_1, v_2)$ in a dataflow graph is an ordered pair of a source actor $v_1 = src(e)$ and sink actor $v_2 = snk(e)$, where $v_1 \in V$ and $v_2 \in V$. When a vertex v executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge. SDF is a specialized form of dataflow where for every edge $e \in E$, a fixed number of tokens is produced onto e every time $src(e)$ is invoked, and similarly, a fixed number of tokens is consumed from e every time $snk(e)$ is invoked. These fixed numbers are represented, respectively, by $prd(e)$ and $cns(e)$. Figure 1-a shows a simple example of a multirate SDF graph (i.e., a graph in which $cns(e)$ and $prd(e)$ differ on one or more edges). Some implementations

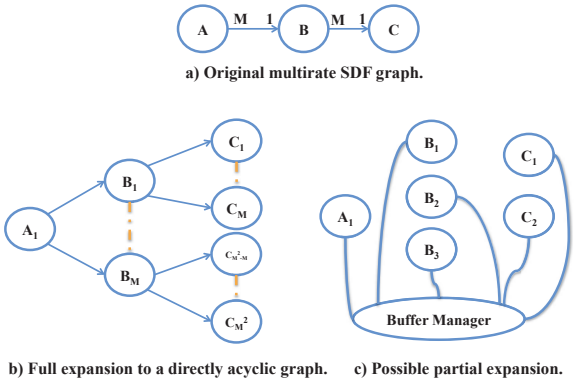


Figure 1. Expansion of a multirate SDF Graph.

of SDF graphs also support non destructive consumption (*peeking*) of an integer number of tokens from an input edge. This integer (non-negative) number is represented by the attribute $peek(e)$ ($peek(e) = 0$ means that peeking is not employed on e). Peek attributes are useful for actors that read histories of tokens such as Finite Impulse Response (FIR) filters. HSDF is a restricted form of SDF where $prd(e) = cns(e) = 1$ for every edge e.

Given an SDF graph G, a *sequential schedule* for the graph is a sequence of actor invocations. A *valid sequential schedule* (*VSS*) guarantees that every actor is fired at least once, there is no deadlock due to token underflow on any edge in the graph, and there is no net change in the number of tokens on any edge (i.e., the total number of tokens produced on each edge during the schedule is equal to the total number consumed from the edge). If a VSS exists for G, then we say that G is *consistent*. For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the number of times that v must be executed in a minimal VSS (i.e., a VSS that involves a minimum number of actor firings) [11]. Such a minimal VSS executes a unit of execution that we refer to as one *iteration* of the given SDF graph. Even though they are formulated in the context of sequential schedules, the concepts of repetition counts and graph iterations are also fundamental for multiprocessor scheduling of SDF graphs.

III. Related Work

Classical multiprocessor scheduling of SDF graphs consists of transforming the input SDF graph to its equivalent HSDF graph or Directed Acyclic Graph (DAG), as shown in figure 1-b. The DAG representation is derived from the HSDF representation by simply removing all edges that have delays (initial tokens) on them. The DAG representation can then be passed to a conventional task graph driven multiprocessor scheduler to generate a schedule for the given SDF graph onto to the available processors [10]. Considerations can be incorporated so that the interprocessor communication costs associated with the removed edges

(edges with delay) are taken into account in construction of the generated schedule (e.g., see [15]).

In [7], architecture and application models are presented to support the Algorithm Architecture Adequation (AAA) methodology. In this methodology, a medium-grain architecture description is applied to capture possible parallelism in the underlying platform. An algorithm description is accepted as a DAG annotated with worst case execution times. Scheduling of the algorithm is performed to satisfy real time constrains without usage of a real-time operating system (RTOS). In [6], the StreamIt compiler is described to utilize different sources of parallelism on coarse-grained multicore architectures. The input application is re-written using the StreamIt language and describes an SDF graph using a subset of SDF modeling semantics. Different graph transformation techniques are explored to amortize the computation to communication ratio for different applications. Data parallelism is achieved by using a static round-robin actor that distributes the loads on the given processors. In [1], a comparison is provided between static and dynamic scheduling. The objective in this comparison is to profile and identify the sources of overhead in a dynamic scheduler that is implemented for a certain model-based programming approach. In contrast, in this paper we develop an implementation framework for dynamic runtime applications that builds on DSP RTOS technology. Potentially, insights from the comparison in [1] can be applied to further improve the performance of our implementation framework.

An adaptive compilation framework that can track dynamic changes in the architecture is presented in [8]. Low overhead runtime compilation is introduced to accommodate possible changes in resources in terms of the number of processors at runtime. The system is originally compiled on a virtual platform using a higher number of cores and an adaptation heuristic is implemented to remap the system onto fewer cores. Online nested task parallelization depending on data set loads is discussed in [16] and [4]. In these works, algorithms are presented to dynamically split an actor and explore more parallelism depending on the actual actor load at runtime. Scheduling adaptive DSP applications such as LTE is presented in [13]. In this work, real time calculation of an efficient schedule is performed at runtime. A full expansion of the dynamic subgraph of the algorithm is suggested. Integration of dataflow semantics with an RTOS is presented in [12].

In comparison to the previous work, we propose partial expansion graphs (PEGs) as a novel dataflow intermediate representation, and foundation to efficiently and effectively schedule static dataflow applications on multiprocessor platforms without fully expanding the SDF graph (into an equivalent HSDF or DAG representation). The proposed strategy makes use of task, data, and pipeline parallelism in the application graph, and allows the designer to easily port legacy code and optimized libraries to RTOS-based parallel implementations. Our strategy allows different scheduling techniques to be implemented to dynamically adapt scheduling decisions when execution times exhibit significant run-time variations. Also, by integrating PEG-driven scheduling with RTOS-based implementation, we allow seamless coexistence of the main DSP application with any secondary tasks that may be developed without use of dataflow models.

IV. Partial Expansion of Dataflow Graphs

In section III, we reviewed the classical method of expanding an SDF graph to generate a corresponding HSDF graph (or closely related DAG) representation, which is suitable for use by multiprocessor schedulers. This technique suffers from two major disadvantages.

First, for a graph that contains multirate edges, the number of generated vertices and edges in the HSDF graph can grow exponentially as shown in figure 1-b, which has M^2+M+1 vertices. If this construction is extended to $n \geq 3$ actors (i.e., as a chain of actors connected by edges having production and consumption rates of M and 1, respectively), then the number of HSDF graph vertices exceeds M^n. Considering the typical number of processors in contemporary platforms, such excessive expansion of the graph representation complicates the job of the multiprocessor scheduler, and may not always lead to better solutions. Second, for applications in which schedules must be recomputed or adapted at run-time, it is difficult to dynamically manage an HSDF representation in terms of memory space and execution time overhead. Such dynamic schedule management is important, for example, when actor execution times exhibit significant run-time variation. However, the overhead of performing such management on an HSDF representation can be prohibitive, thereby forcing use of inflexible static schedules that are not matched to execution time dynamics, and perform poorly.

Intuitively, a *partial expansion graph* (*PEG*) $G_p = (V_p, E_p)$ is an undirected graph that is used for scalable mapping of an SDF graph G onto a programmable platform. The vertex set $V_p = I_p \bigcup \{B\}$, where I_p corresponds to a set of *instances*, which execute groups of successive actor firings, and B is a special inter-vertex coordination actor called the *buffer manager* of the PEG. The edge set of the PEG is defined as $E_p = \{\{B, i\} \mid i \in I_p\}$ — in other words, an (undirected) edge is connected between the buffer manager and every other vertex in the PEG.

Each actor v in G corresponds to a set I_v of N_v instances in G_p ($I_v \subset I_p$), where the mapping $f_v : V \rightarrow N_v$ can be viewed as a design parameter of the PEG. Given a PEG, and a vertex v in the corresponding SDF graph G, N_v is referred to as the *instantiation factor* of v. In general, N_v is positive-integer-valued. However, if v has *state* (internal actor variables whose values persist across firings), then N_v is always equal to 1.

Figure 1-c shows one possible partial expansion graph for the SDF graph of Figure 1-a. In this example, the original actors A, B and C, which satisfy $q(A) = 1$, $q(B) = M$, and $q(C) = M^2$, are expanded in G_p to $N_A = 1$, $N_B = 3$ and $N_C = 2$ different groups of instances. Such expansion (or consolidation, when viewed in terms of the underlying HSDF graph) could be done, for example, for reasons related to load balancing, as discussed later in the paper.

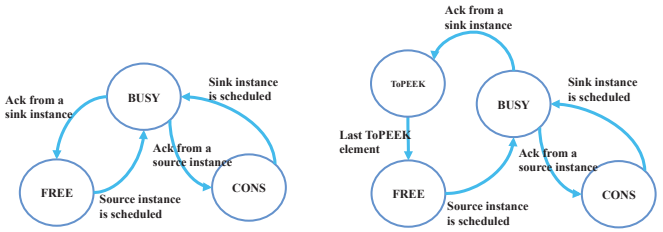
In contrast to the original SDF graph G, the PEG representation relates the actors and edges in G to a particular multiprocessor implementation. The set of instances that correspond to a given actor in the PEG can execute concurrently on the available processors. Therefore, in a platform with C cores where every instance is mapped to a different core, we will have $N_v \leq C$. Associated with every PEG instance, there is a unique kernel (software code block) that executes the code provided with the corresponding SDF actor v. A call to this kernel is called an *activation* of the associated PEG instance. An activation can be viewed as a vectorized firing of v, through the kernel associated with i, that consumes and produces an amount of data equal to an integer multiple of the consumption and production rates of the input and output edges, respectively, for v. This integer multiple is called the *vectorization factor* of the associated activation. Thus, an activation executes a number of successive actor firings that is equal to its vectorization factor. Use of vectorized firings for SDF graphs has been studied extensively, starting with the foundational work of Ritz, Pankert and Meyr [14]. Our PEG formulation provides a novel framework in which vectorization can be integrated efficiently into multiprocessor scheduling contexts.

In the next section, we discuss the special buffer manager vertex B, which is a critical component in the definition and use of the PEG model.

V. Buffer Manager

When implementing a PEG, the buffer manager vertex B is mapped into a software process that coordinates the sharing of state across instances that share the same SDF graph actors, and coordinates data transfer between communicating instances that are mapped to different processors. Within this process, there is a local state that stores information corresponding to the edges in the enclosing PEG G_p, and special data packets, called *PEG messages*, that are associated with instances in G_p. A PEG message for a PEG instance i encapsulates one more pointers to memory blocks that implement the SDF graph edges incident to $A(i)$, where $A(i)$ denotes the SDF graph actor that corresponds to i.

An instance i is activated when it receives a PEG message that contains a number of pointers equal to the of number edges that $A(i)$ is connected to. These pointers have along with them the required information for the positions and amounts of data to be consumed from the referenced input buffer(s), and the data to be produced onto the referenced output buffer(s). A PEG message is sent to the processor that contains an instance to schedule its execution once all buffers associated with the instance are ready, as explained in the next section. Once the instance finishes execution, it acknowledges B by sending back the same PEG message. For interprocessor communication across instances, we assume in this paper that buffers in shared memory are used.



a) Buffers manager FSM for zero peek edges.

b) FSM to support the peek attribute in shared memory architectures.

Figure 2. Finite state machines for buffer slot states.

During application initialization, the buffer manager allocates the required amount of memory (i.e., based on the buffer length) for each buffer that connects a communicating pair of instances. These buffer lengths can be determined statically based on classical SDF analysis techniques (e.g., see [15]).

In our model of PEG-based implementation, buffers are decomposed into *slots*, where each slot is large enough (in terms of number of bytes) to store some number of tokens that is less than or equal to the corresponding buffer size (i.e., the maximum token capacity of the buffer). Associated with every slot s, there is a *slot status* variable $\nu(s)$, which helps to select among the possible actions that can be performed on the slot. In our implementation, $\nu(s)$ can have four possible values, which are summarized as follows.

- FREE state: The bytes that correspond to the slot are free to be overwritten by an instance that produces tokens onto this buffer.
- CONS state: The bytes that correspond to the slot are ready to be read by an instance that consumes tokens from this buffer.
- BUSY state: The underlying bytes are being either produced onto or consumed from by an instance. This acts like a semaphore on the slot to help avoid race conditions.
- ToPEEK state: If the value of the peek attribute (see Section II) of an edge is greater than zero, this status is given to a slot after tokens from it have been consumed. This allows subsequent activations to non-destructively read the corresponding tokens if they fall within the range to be peeked during an activation.

Figure 2-a illustrates the finite state machine (FSM)

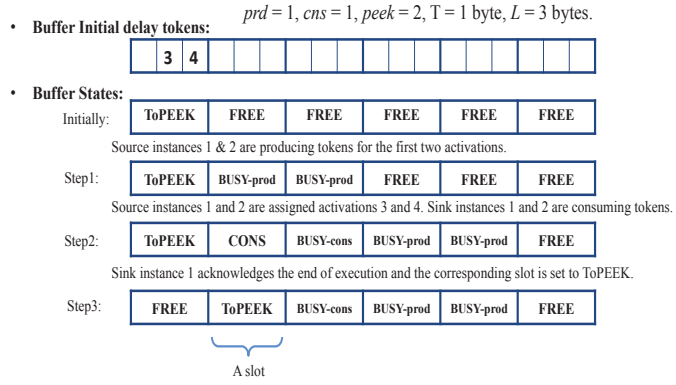


Figure 3. Updates of buffer states during graph execution.

that the buffer manager implements to change the states of slots for edges with zero-valued peek attributes. For a delayless edge, all slots are originally in the FREE state. The buffer manager starts assigning buffer space to source instances and marks the corresponding slots as BUSY. As the buffer is initially empty, the sink actors will be inactive. Upon reception of acknowledge messages that confirm the completion of source activations, the buffer manager sets the produced slots to the CONS state. If the number of slots that have the CONS status is enough to trigger sink instances, the buffer manager sets them to BUSY and activates the sinks. Finally, upon the completion of the sinks, the consumed slots are returned back to the FREE state. An extended version of the buffer manager FSM, which supports the peek attributes of SDF graph edges, is shown in Figure 2-b.

VI. Slot Size Selection

The slot size L interacts with key performance factors including run-time buffer manager overhead, cache performance, and allowable vectorization factors. Simulation of alternative slot sizes within the structured framework of PEG-based implementation can help to determine an efficient slot size for a given application/platform combination. Figure 3 illustrates a simulation for executing the FSM in Figure 2-b for a slot size of $L = 3T$, where T is the token size (in bytes). Important to notice is that the *peeking_pointer* p (that points to the last peeked token) may not always be aligned with the beginning or end of a slot. In this case, the entire slot will be in the ToPEEK status while p is pointing into the slot, and the slot will be cleared to the FREE state only after p falls outside the range of tokens covered by the slot.

The slot size L can be useful also in the modeling of application/platform interactions that involve caches. In cache enabled processors, such as the Texas Instruments C64X+ family of digital signal processors, operations to load, invalidate and write back memory segments can only take place in terms of cache line sizes (e.g., 128 bytes). This means that the memory that corresponds to a cache line cannot be simultaneously accessed by two processors, otherwise a race condition can occur. This problem can be solved if L is chosen to be a multiple of the cache line size. To guide efficient and correct buffer manager implementation, this cache-based condition can be applied in conjunction with the simulation based analysis described above.

VII. Dynamic Scheduling

The buffer manager B schedules an activation only if the corresponding input and output dataflow resources are available. Initially, all of the buffer states are checked and if all ports connected to an instance i are ready, then B sends a PEG message to schedule the activation of i on its local processor. Once the activation completes execution, the instance i sends an acknowledgment message to B. The buffer manager then updates the states of all of the buffers connected to i, and triggers subsequent instances for which all input and output buffers are ready.

From the discussion in Section IV, instances can be viewed as threads that execute activations. In this context, we distinguish between two scheduling sub-tasks: *mapping* an instance i to be executed on a processing node p, and *assigning* and activation a to be run by an instance i. These decisions highly affect the final system performance in terms of latency and throughput, and can take place either at compile time or run-time. Also, instances assigned to the same processor can interrupt one another depending on their priorities. Intuitively, the mapping functions determine the maximum load on every processor as the application executes. The separation of mapping and assignment decisions gives the system designer the flexibility to either statically control the system behavior, or postpone some scheduling decisions to run-time. In the experiments reported on in this paper, we fix the mapping decisions at compile time, and we compare between a) taking the assignment decisions at compile time using classical methods, and b) taking the assignment decisions dynamically at run-time.

Low priority instances can be preempted by higher priority ones if the resources required by the higher priority instances become available. Since the buffer manager always checks for available resources before activation, deadlock is systematically avoided. As execution of a PEG-driven SDF graph implementation proceeds, both data and task parallelism can be exploited, depending, respectively, on whether instances of the same actor or instances of different actors execute at the same time. These two forms of parallelism in general reduce the total latency of the application; however, they may not be sufficient to utilize all of the processors, and some amount of pipeline parallelism (parallelism across distinct graph iterations) may also need to be exploited.

We have developed Algorithm 1 to dynamically assign activations to instances of a given actor. Upon receipt of a PEG message, it will schedule the next activation a to

execute if the local processor is idle or if a has higher priority compared to the currently executing thread. The proposed heuristic uses a First Acknowledge First Assign (FAFA) mechanism for the buffer manager B to assign activations to instances. This mechanism is based on the assumption that the instance that finishes its execution first is ready to receive a new activation.

In Algorithm 1, the "instances allocation variable" *i_alloc* stores the last activation value that is assigned to every instance. The buffer manager later checks this value to construct the PEG message to be sent when an activation is ready. The utility of this scheduling approach is demonstrated in our experiments presented in Section IX.

Algorithm 1 FAFA dynamic scheduling heuristic.

```
{Initialization: Distribute the activations act of an actor
across the available instances}
for all Actors v ∈ V do
  for all Instances i in N_v do
    i_alloc[v][i] ← i;
  end for
  last_act[v] ← N_v;
end for
{Upon reception of a PEG msg}
for all Buffer states b connected to that actor do
  if b is input to i then
    if msg.act == i_alloc[b.src_actor][msg.i_id] then
      i_alloc[b.src_actor][msg.i_id] ←
          ++last_act[b.src_actor];
    end if
  end if
  if b is output to i then
    if msg.act == i_alloc[b.snk_actor][msg.i_id] then
      i_alloc[b.snk_actor][msg.i_id] ←
          ++last_act[b.snk_actor];
    end if
  end if
end for
```

VIII. Code Generation

Our proposed PEG-based implementation methodology can be realized on a platform consisting of multiple processors that have an RTOS running on them. The basic implementation model assumes that actor instances and the buffer manager will run within threads. The operating system must also provide mechanisms to allocate shared memory space, perform IPC to send and receive PEG messages, and schedule threads on individual cores.

We have developed a code generator to automate the realization of a multicore DSP software implementation from a PEG-based schedule specification and implementations of the underlying SDF graph actors. Input to the code generator consists of a library of algorithm kernels that represent actors, a scheduling solution in terms of buffer lengths, amounts of expansion (numbers of PEG instances) for the actors, instance-to-processor mappings, instance priorities, and required implementation attributes (e.g., filter coefficients and values for initial tokens). The output of the code generator is then a complete multicore software realization of the given SDF graph using the available Application Programming Interfaces (APIs) associated with the targeted RTOS and processing platform. In this paper, we implemented our PEG realization using the multithreading and IPC APIs provided by the Texas Instruments DSP/BIOS RTOS. The output of our developed code generator is in the form of embedded C code, which can be compiled using the Texas Instruments C6000 compiler and linker. For every actor in the system, a header file is generated, which contains information about the number and types of edges that are connected to the actor in the form of integer identifiers.

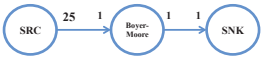
An initialization function and an execution function are generated for every instance in the system. The initialization function for an instance i runs at system start-up to set up a "mailbox" (e.g., MessageQ in DSP/BIOS) for every other instance from which i can receive PEG messages. Upon reception of a PEG message, the thread that wraps an activation is scheduled. This wrapper is generated as the execution function for the associated instance. Within this function, the PEG message is read from the MessageQ, the actor kernel is called, and upon completion of the actor kernel, the PEG message is acknowledged back to the buffer manager.

After parsing the given PEG-based scheduling solution, the code generator constructs a folder that contains the required .c, .h, and configuration files for every core in the target platform. The generated multi-core implementation can then be compiled and run for profiling or validation.

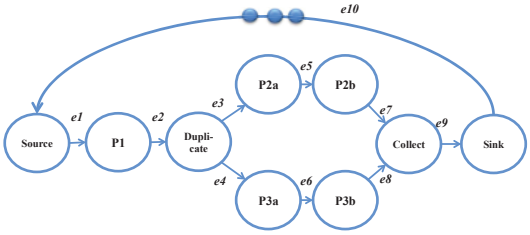
IX. Evaluation

We have implemented and experimented with the PEG strategy by using various SDF-based benchmarks as DSP applications and the Dataflow Interchange Format (DIF) [9] for dataflow graph specification. The benchmarks were constructed to independently profile different PEG aspects, such as speedups, dynamic scheduling characteristics, and buffer manager overhead. We executed all of our experiments on a Texas Instruments TMS320C6472 six-core DSP device with 769K-Byte shared memory and 608 K-Bytes of private, configurable L2 memory or cache on each core. In these experiments, the buffer manager is implemented on a separate core, while the other five cores implement the other PEG graph vertices (actor instances).

We carried out three experiments. For the first experiment, we implemented the "pseudo-communication graph" shown in Figure 4-b. This application is representative of the front end processing structure for a digital communication



a) Intrusion detection application using the Boyer-Moore actor.



b) Pseudo-communication receiver application.

Figure 4. Evaluation benchmarks.

receiver. Input data is received by the actor *Source*. Actor P_1 represents a filter and has a positive-valued peek attribute. The two parallel branches represent possible processing on the I-Q channels and the last two actors, *Collect* and *Sink*, represent the application back-end. The focus of this experiment is to measure the speedups for different sources of parallelism using the PEG strategy — therefore, the actor loads (execution times), and the production and consumption rates of different edges are manually adjusted to generate different graphs.

Figure 5 shows possible speedups for different application graph configurations that are derived from the pseudo-communication synthetic "benchmark template" shown in Figure 4-b. In the *Task graph*, the execution time loads of the two parallel branches represent 44% of the total execution time load. However, using only task parallelism is not sufficient to achieve a reasonable speedup, while mapping different instances of the pipeline to different cores and combining both task and pipeline parallelism gives a significantly better speedup of 4.15x. The *Data graph* illustrates use of the PEG strategy to avoid full expansion. In this application model, the production and consumption rates in the graph are adjusted such that actor P_1 has a repetition count of 10. However, as there are only 5 cores in the platform, P_1 is partially expanded 5 times and its load constitutes 81% of the total graph load. In this example, a speedup of 4.2x over single core implementation is achieved by using the three different levels of parallelism. Finally, in the *Pipeline graph*, all actors have the same load and good speedups can be achieved only using pipeline parallelism. As there are 9 instances in total, the maximum theoretical speedup is 4.5x and our implementation achieves a speedup of 3.84x.

Our second experiment is summarized by Figure 6, which quantifies how the PEG strategy can be useful to achieve speedups for different computation-to-scheduling ratios. This experiment is carried out on the *Pipeline Graph*

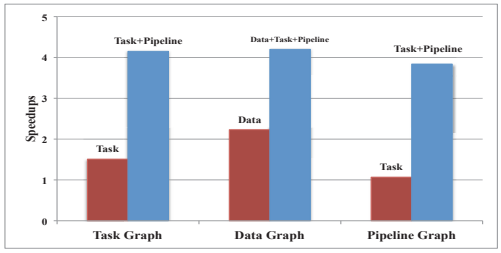


Figure 5. Speedups for different sources of parallelism using the PEG strategy.

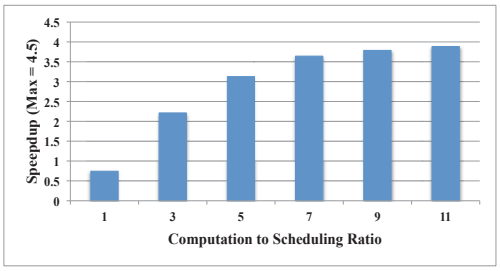


Figure 6. Efficiency of the PEG strategy for different computation to scheduling ratios.

described above. In this experiment, all of the application actors have the same execution time load λ, which is assumed to be a multiple of the scheduling load S. Here, by the "scheduling load", we mean the average number of cycles spent in the buffer manager to schedule one activation. The horizontal axis represents the ratio of λ to S for different scenarios. For this experiment, we see that for a ratio of 7:1 (or greater), reasonable speedups can be achieved. For lower ratios, the granularity of computation is too small to adequately amortize the scheduling overhead.

Our third experiment, summarized in Figure 7, shows a comparison between the FAFA algorithm for PEG-based dynamic scheduling and the conventional round robin distribution of activations to instances. In this graph, we applied the Boyer-Moore string matching actor shown in Figure 4-a [5]. This actor can be used in many important DSP applications — e.g., in network intrusion detection, where incoming packets are searched for sets of malicious strings.

The runtime of the Boyer-Moore actor can exhibit significant variation across different packets — e.g., depending on the target string position when a match is found. The maximum standard deviation in runtime is achieved when the matching string can be located in any position within the packet, and zero standard deviation means that all matching strings are located in the middle of the packet. The horizontal axis shows the normalized standard deviation for a single activation's runtime compared to the maximum standard deviation. Within the enclosing SDF graph, the Boyer-Moore actor has a repetition count of 25, but it is partially expanded on only 5 cores due to platform limitations. The empirical

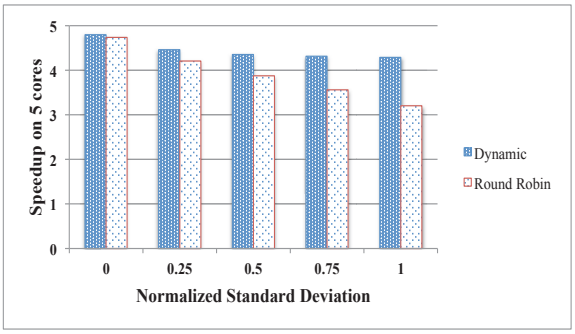


Figure 7. Comparison between round robin and dynamic scheduling of activations.

results for this "embarrassingly parallel" but "unpredictable-load" application show that both scheduling algorithms achieve the same speedup when there is no variability in runtime. However, the PEG-based dynamic scheduler is 35% more efficient than conventional methods when execution times exhibit high variability.

X. Conclusion

In this paper, we have presented a new intermediate model and associated implementation strategy called the partial expansion graph (PEG). The PEG overcomes conventional problems associated with exponential growth of SDF graph expansions, allowing parallelism to be exposed and exploited judiciously based on levels that match reasonably to the target platform. We have shown that significant speedups on a state-of-the-art multicore DSP platform can be achieved using the proposed PEG methodology, and demonstrated higher speedups, compared to classical round robin scheduling, by using PEG-based dynamic scheduling techniques. We have also presented experimental analysis that quantifies various trade-offs associated with PEG-based implementation, and discussed integration with off-the-shelf RTOSs. Useful directions for future work include optimization of buffer sizes under PEG implementation, use of hardware support for buffer manager implementation, and development of automatic scheduling techniques that target PEG-based implementation.

References

[1] O. Arnold and G. Fettweis. On the impact of dynamic task scheduling in heterogeneous mpsocs. In *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, July 2011.

[2] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, 2010.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[4] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2008.

[5] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In *ACM-SIAM Symposium on Discrete Algorithms*, 1991.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.

[7] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, 2003.

[8] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[9] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[10] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Journal of the Association for Computing Machinery*, 31(4):406–471, December 1999.

[11] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, February 1987.

[12] Y. Oliva, M. Pelcat, J. Nezan, J. Prevotet, and S. Aridhi. Building a RTOS for MPSoC dataflow programming. In *Proceedings of the International Symposium on System-on-Chip*, 2011.

[13] M. Pelcat, J. Nezan, and S. Aridhi. Adaptive multicore scheduling for the LTE uplink. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2010.

[14] S. Ritz, M. Pankert, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[15] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.

[16] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming*, 2010.