In Proceedings of the International Workshop on Software and Compilers for Embedded Processors, Dallas, Texas, September 2005.

# Software Synthesis from the Dataflow Interchange Format

Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, USA
{jerryhsu, myko, ssb}@eng.umd.edu

## ABSTRACT

Specification, validation, and synthesis are important aspects of embedded systems design. The use of dataflow-based design environments for these purposes is becoming increasingly popular in the domain of digital signal processing (DSP). The dataflow interchange format (DIF) [11] and the associated DIF package have been developed for specifying, working with, and transferring dataflow-based DSP designs across tools. In this paper, we present the newly developed *DIF-to-C software synthesis framework* for automatically generating monolithic C-code implementations from DSP system specifications that are programmed in DIF. This framework allows designers to efficiently explore the complex range of implementation trade-offs that are available through various dataflow-based techniques for scheduling and memory management. Furthermore, the DIF-to-C framework provides a standard, vendor-neutral mechanism for linking coarse grain dataflow optimizations with fine grain hand-optimized libraries and the large body of optimization techniques in the area of C compilers for DSP. Through experiments involving several DSP applications, we demonstrate the novel and useful capabilities of our DIF-to-C software synthesis framework.

## Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications — Data-flow languages; D.2.2 [**Software Engineering**]: Design Tools and Techniques — Computer-aided software engineering (CASE).

## General Terms

Design, Languages.

## Keywords

Software synthesis, Dataflow Interchange Format, DIF.

## 1. INTRODUCTION

Modeling digital signal processing (DSP) applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models (e.g., see [1,6,15,17]) have been developed for different aspects of DSP design [3]. A growing set of commercial and research-oriented tools incorporate dataflow semantics, including ADS from Agilent [23], the Autocoding Toolset from MCCI [18], CoCentric System Studio from Synopsis [7], Compaan from Leiden University [19], Gedae from Gedae Inc., Grape from K. U. Leuven [14], LabVIEW from National Instruments, PeaCE from Seoul National University [20], Ptolemy II from U. C. Berkeley [8], and StreamIt from MIT [21].

Most dataflow-based DSP design tools provide intuitive graphical design environments, libraries of functional modules, and capabilities for simulating algorithm specifications based on synchronous dataflow (SDF) [15] or closely related models. Moreover, some tools are also capable of synthesizing hardware and/or software implementations. However, there is great variation across the front- and back-ends of these tools: these tools generally use different specification formats, provide different sets of DSP libraries, and target different embedded processing platforms. Even though, developing or migrating DSP designs across multiple tools often becomes desirable because they may have complementary features (e.g., simulation vs. synthesis, hardware vs. software support, etc.), and different generations of designs may be best suited to different tools.

A critical problem arises in transferring a DSP design across dataflow-based tools due to the lack of a vendor independent language for specifying DSP-oriented dataflow graphs. The dataflow interchange format (DIF) [10,11] and the associated DIF package, a Java package that provides dataflow representations and algorithm implementations, have been developed for specifying and working with DSP applications across the evolving family of dataflow-based DSP design tools. Recently, a DIF-based porting approach has been developed for porting DSP designs across dataflow-based tools [9]. Figure 1 illustrates the methodology of using DIF to interface abstract dataflow models, dataflow-based DSP design tools, and their supported families of embedded processing platforms.

The automation of software synthesis from DIF specifications to C programs is a very useful feature that we have recently integrated into DIF. Since the DIF language is designed as a programming language as well as an interchange format (in particular, it is designed to be read and written intuitively by designers, not just to be generated and parsed by tools), software synthesis capability in DIF provides a new path to implementation from standalone use of the DIF package, in addition to implementation through tools. With this software synthesis framework, designers need only to specify the desired dataflow graph topology and hierarchy, the relevant dataflow attributes (production and consumption rates, delays, etc.), and actor attributes (function associations, edge/port connections, parameters, etc.). Then C code that implements the

dataflow specification is generated automatically. Figure 1 highlights the DIF-to-C framework in the DIF methodology. Since most programmable digital signal processors (PDSPs) and other types of embedded processors provide C compilers, and furthermore, many PDSP vendors and third-party companies provide hand-optimized C libraries, the DIF-to-C framework offers a valuable link between formal, domain-specific DSP design and processor/platform-specific implementation infrastructure.

The DIF-to-C framework provides software synthesis capability that operates through the following *unique* integration of important features. 1. *Extensibility*: This software synthesis framework is library-based (in contrast to pre-defined in-line code generation) such that DIF programmers can associate actors with desired C functions either designed by themselves or obtained from any existing library. It supports general C-based libraries, e.g., one-dimensional signal processing libraries and image/video processing libraries from Texas Instruments [24,25], and can easily be extended to support C-based APIs, such as VSIPL [12]. 2. *Flexibility*: The DIF package provides representations of various dataflow models and efficient implementations for many scheduling algorithms and buffering techniques. This large and growing set of models, algorithms, and techniques spans a broad range of the design space: designers can easily explore different combinations and determine trade-offs among key metrics such as code size, memory requirements, and performance. 3. *Portability*: A systematic porting mechanism is available in DIF that enables efficient migration of designs across dataflow-based DSP design tools [9]. By integrating the DIF-to-C framework with the porting mechanism, a DIF specification of a design can be automatically implemented on various embedded processing platforms through the software synthesis capability as well as through the supported design tools.

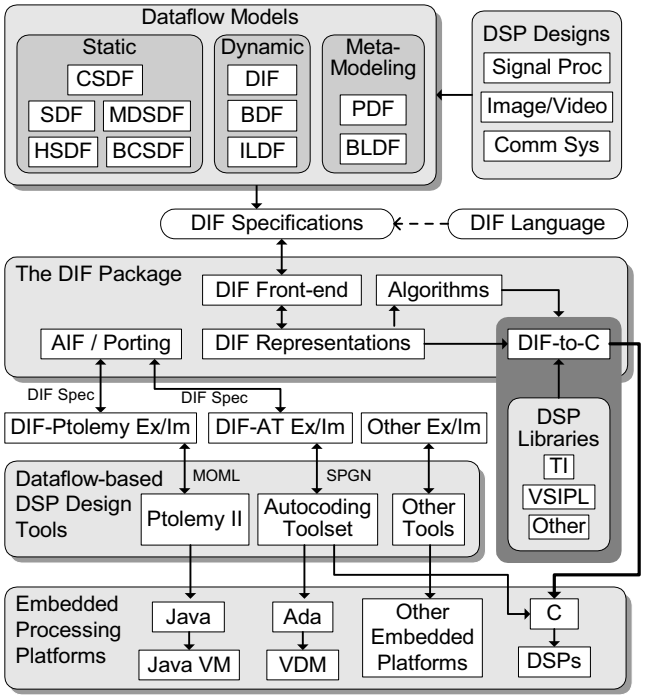



**Figure 1. The role of DIF in DSP system design.**

Figure 2 illustrates the design flow that underlies the DIF-to-C software synthesis framework. The organization of this paper follows this design flow. In Section 2, we describe the programming phase. We then introduce the compilation phase, including scheduling algorithms in Section 3, and buffering techniques in Section 4. Next, we present the code generation phase in Section 5, and demonstrate simulation results of various synthesized DSP applications in Section 6. We conclude in the final section.

## 2. DATAFLOW MODELING AND DIF

The programming phase of the design flow, as shown in Figure 2, includes modeling a given DSP application based on dataflow modeling principles, and specifying it through the DIF language. We currently support SDF in the DIF-to-C framework. Extending software synthesis support in DIF to other dataflow modeling semantics is an active area of ongoing work.

### 2.1 Dataflow Modeling

In the dataflow modeling paradigm, computational behavior is depicted as a dataflow graph. A dataflow graph $G$ is an ordered pair $(V, E)$, where $V$ is a set of nodes, and $E$ is a set of directed edges. A dataflow graph node (actor) $v \in V$ represents a computational module. A directed edge $e = (v1, v2)$ represents a buffer from its source node $v1 = \text{src}(e)$ to its sink node $v2 = \text{snk}(e)$ and imposes precedence constraints for proper scheduling of the dataflow graph. Dataflow graphs naturally capture the data-driven property in DSP computations. An actor $v$ can fire (execute) only when it has sufficient tokens on all of its incoming edges $\text{in}(v) = \{e \in E | \text{snk}(e) = v\}$. When firing, it consumes certain numbers of tokens from its incoming edges, executes its associated computation, and produces certain numbers of tokens on its outgoing edges $\text{out}(v) = \{e \in E | \text{src}(e) = v\}$.

For a sophisticated DSP application, the overall system is usually modeled as a hierarchical graph in which the computations associated with certain actors, called *hierarchical actors*, can be speci-

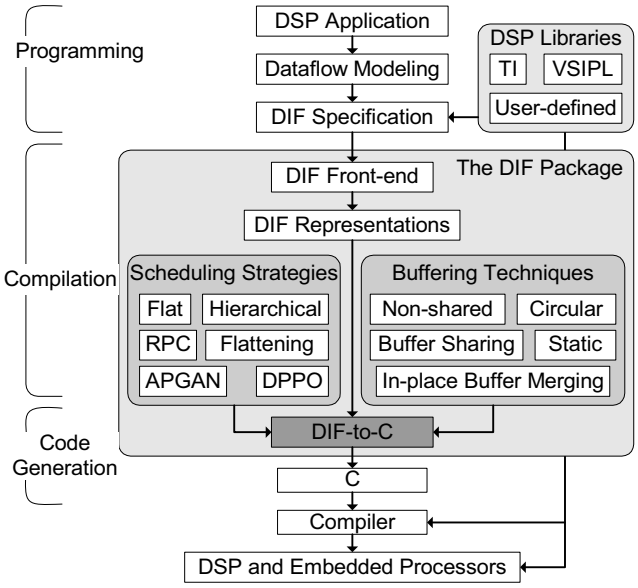



**Figure 2. The design flow of the DIF-to-C framework.**

fied as nested dataflow graphs. The formal dataflow graph definition described above is insufficient to represent such hierarchical nesting. Therefore, a *hierarchy* structure is introduced in DIF for specifying hierarchical dataflow graphs. In DIF semantics, a node can represent either an indivisible computation or a hierarchically-nested subgraph (called a *supernode* in DIF). A DIF hierarchy $H = (G, I, M)$ consists of a graph $G$ with an interface $I$ and a mapping $M$. Suppose that a supernode $s$ in $G$ represents a nested subhierarchy $H' = (G', I', M')$, then a refinement $H' = \text{subhrcy}(s)$ is established for refining $s$ to $H'$. The mapping $M$ can be described as a function whose domain is simply the set of supernodes in $G$ and whose range is obtained through the property $M(s) = \text{subhrcy}(s)$ for every supernode $s$. A directed port $p$ of the hierarchy $H$ is a dataflow gateway through which tokens (objects that encapsulate data values) flow into (input port) or flow out of (output port) the graph $G$. The interface $I$ is a set consisting of all ports of $H$. Viewed from within $G$, a port $p \in I$ associates with a node $v$ in $G$, and this is denoted as $v = \text{assoc}(p)$. Suppose that $H$ is a subhierarchy represented by a supernode $s''$ in a higher level graph $G''$, i.e., $H = \text{subhrcy}(s'')$. Then viewed from outside $G$, a port $p \in I$ can either connect to an edge $e''$ in the higher level graph $G''$ or connect to a port $p''$ in the higher level hierarchy $h'' = (G'', I'', M'')$; these are denoted as $e'' = \text{connect}(p)$ or $p'' = \text{connect}(p)$, respectively.

## 2.2 Synchronous Dataflow

In synchronous dataflow (SDF) [2,15], an edge $e$ represents a FIFO queue and can have a non-negative integer delay, denoted as $\text{delay}(e)$, associated with it. Each delay unit is functionally equivalent to a $z^{-1}$ operator and represents an initial token queued on $e$. The number of tokens produced/consumed on $e$ by a firing of $\text{src}(e)/\text{snk}(e)$ is restricted to be a constant positive integer known at compile time. These numbers are referred to as the production rate and consumption rate of $e$ and are denoted as $\text{prd}(e)$ and $\text{cns}(e)$, respectively. SDF possesses useful compile-time capabilities of deadlock detection, bounded memory determination, and static scheduling, and is highly suitable for modeling multi-rate DSP systems [3].

Nested hierarchical SDF graphs are constructed in DIF based on the concepts reviewed in Section 2.1. For a node $v$ associated with an output/input port $p$, the production/consumption rate of that connection is denoted as $\text{prd}(p)/\text{cns}(p)$, since the edge in that connection is outside the graph. Furthermore, because production/consumption rates of a supernode depend on the repetition vector of the subgraph, they are left unspecified and are computed during the compilation phase.

## 2.3 The Dataflow Interchange Format

As mentioned earlier, the Dataflow Interchange Format is a language for specifying mixed-grain dataflow models for DSP systems. The current version of the DIF language is capable of representing systems that are based on a variety of different dataflow modeling formats [10], e.g., SDF [15], CSDF [6], and PSDF [1]. Figure 3 sketches the DIF language syntax. For a detailed description of the DIF language and the language grammar, we refer the reader to [10]. DSP applications specified by the DIF language are usually referred to as *DIF specifications*. Figure 4 presents a tree-structured filter bank modeled in SDF. Here, supernodes are shown in bold blocks, and production and consumption rates are indicated at the ends of edges and alongside ports. The corresponding DIF specification is shown partially to illustrate the process of programming in DIF.

Given a DIF specification, the first step in the compilation phase is to extract the underlying dataflow graph in order to construct the corresponding *DIF representation*, the instances of dataflow graph classes realizing a dataflow specification in the DIF package. As shown in Figure 1 and Figure 2, the DIF package provides the DIF front-end tool for automatically converting between DIF specifications and DIF representations. For a detailed description of the DIF front-end and the DIF representation, we refer the reader to [10].

# 3. SCHEDULING

In the compilation phase, we compute a *schedule* of the dataflow graph through one of various algorithm implementations that operate on the internal DIF representations. Here, by a schedule, we mean a sequence of actor firings or more generally, any sequencing mechanism for executing actors (including static, dynamic, and hybrid static/dynamic sequencing). Since the DIF-to-C framework is presently based on SDF semantics, we focus in the remainder of this paper on schedules that involve purely static sequencing, which are most natural for implementation from SDF graphs.

There is a complex range of trade-offs involved during the scheduling phase, and these trade-offs heavily affect all key implementation metrics. One of the benefits of using the DIF-to-C framework is that a significant variety of scheduling algorithms is available in

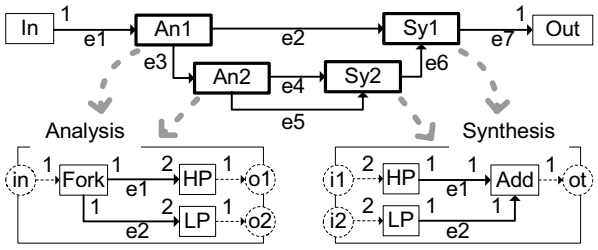
```
dataflowModel graphID {
  basedon { graphID; }
  topology {
    nodes = nodeID, ...;
    edges = edgeID (srcNodeID, snkNodeID), ...; }
  interface {
    inputs = portID [:nodeID], ...;
    outputs = portID [:nodeID], ...; }
  parameter {
    paramID [:datatype];
    paramID [:datatype] = value;
    paramID [:datatype] : range; }
  refinement {
    subgraphID = supernodeID;
    subPortID : edgeID;
    subParamID = paramID; }
  builtInAttr {
    [elementID] = value;
    [elementID] = id;
    [elementID] = id1, id2, ...; }
  attribute usrDefAttr{
    [elementID] = value;
    [elementID] = id;
    [elementID] = id1, id2, ...; }
  actor nodeID {
    computation = stringValue;
    attrID [:attrType] = value;
    attrID [:attrType] = id;
    attrID [:attrType] = id1, id2, ...; }
}
```

**Figure 3. An overview of the DIF language syntax.**

the DIF package, and this provides great flexibility for exploring trade-offs.

## 3.1 SDF Scheduling Preliminaries

An SDF graph $G = (V, E)$ has a valid schedule (is *consistent*) if it is free from deadlock and is sample rate consistent (i.e., it has a periodic schedule that fires each actor at least once and produces no net change in the number of tokens on each edge) [2]. $G$ is *sample rate consistent* if there is a positive integer solution to the *balance equations*: $\text{prd}(e) \times \boldsymbol{q}[\text{src}(e)] = \text{cns}(e) \times \boldsymbol{q}[\text{snk}(e)]$, $\forall e \in E$. The minimum solution for the vector $\boldsymbol{q}$ is called the *repetition vector* of $G$, and $\boldsymbol{q}[v]$ (where $\boldsymbol{q}$ henceforth represents the minimum solution of the balance equations) is called the *repetition count* of actor $v$. A valid schedule is then a sequence of actor firings where each actor $v$ is fired $\boldsymbol{q}[v]$ times and the firing sequence obeys the precedence constraints imposed by the SDF graph.




```
sdf Analysis1 {
  topology {
    nodes = Fork, HP, LP;
    edges = e1 (Fork, HP), e2 (Fork, LP);          }
  interface {
    inputs = in : Fork;
    outputs = o1 : HP, o2 : LP;          }
  production { e1 = 1; e2 = 1; o1 = 1; o2 = 1; }
  consumption { e1 = 2; e2 = 2; in = 1; }
  attribute datatype {e1="float"; e2="float"; in="float"; ...}
  actor HP {
    computation = "FIR";
    decimation = 2;
    interpolation = 1;
    coefs = [...];       }
  ...   }
sdf Analysis2 { basedon {Analysis1;} }
sdf Synthesis1 {...}
sdf Synthesis2 { basedon {Synthesis1;} }
sdf filterBank {
  topology {
    nodes = In, An1, An2, Sy1, Sy2, Out;
    edges = e1 (In, An1), e2 (An1, Sy1), ..., e7 (Sy1, Out); }
  refinement {Analysis1=An1; in:e1; o1:e2; o2:e3; }
  refinement {Analysis2=An2; in:e3; o1:e4; o2:e5; }
  refinement {Synthesis1=Sy1; i1:e2; o2:e6; ot:e7;}
  refinement {Synthesis2=Sy2; i1:e4; o2:e5; ot:e6;}
  ...
  production {e1 = 1;}
  consumption {e7 = 1;}
  attribute datatype { e1 = "float"; ...; e7 = "float"; }
  ... }
```

**Figure 4. A filter bank graph and the DIF specification.**

To save code space, actor firing sequences can be incorporated within looping constructs. A *schedule loop*, $L = (nT_1T_2\ldots T_m)$, represents the successive repetition $n$ times of the invocation sequence $T_1T_2\ldots T_m$, where each $T_i$ is either a firing or a (nested) schedule loop. A *firing*, $nv$, represents the firing of an actor $v$ $n$ times in succession. Informally, a looped schedule $S$ is an SDF schedule that is expressed in terms of schedule loop notation, and therefore can contain one or more looping constructs. If every actor appears only once in $S$, $S$ is called a *single appearance schedule* (SAS), otherwise, $S$ is called a *multiple appearance schedule* (MAS). Looped schedules provides compact representation for actor firing sequences and enables efficient implementation through code generation.

Furthermore, any SAS for an acyclic SDF graph can be represented in the *R-schedule* form [2], which can be naturally represented as a schedule tree. A *schedule tree* is in turn a binary tree where an internal node represents a subschedule and a leaf node represents a firing. It provides a convenient internal representation for SDF scheduling, and is widely used in computing schedules and buffer minimization. An example of an R-schedule and the corresponding schedule tree is shown in Figure 8.

## 3.2 Scheduling Algorithms for SDF Graphs

Given a schedule $S$ of an SDF graph $G$, the *buffer size* required for each edge $e$ is the maximum number of tokens simultaneously queued on $e$ during an execution of $S$, denoted as $\text{maxToken}(e, S)$. The *total buffer requirement* for executing $S$, denoted as $\text{buffer}(G, S)$, is the sum of $\text{maxToken}(e, S)$ of every edge $e$ in $G$.

In general, the problem of computing a buffer-optimal SDF schedule is NP-complete, and furthermore, buffer-optimal schedules are usually MASs whose lengths generally increase exponentially in the size of the SDF graph. An SAS is often preferable due to its optimally compact implementation containing only a single copy of code for every actor. A valid SAS exists for any consistent and acyclic SDF graph and can be easily derived from a *flat scheduling strategy*, i.e., a strategy that computes a topological sort of the SDF graph and iterates each actor $v$ $\boldsymbol{q}[v]$ times. The flat strategy is useful because it is simple, fast, and results in schedules with relatively low context switching. However, the flat strategy may also lead to relatively large buffer requirements and latencies.

Several scheduling algorithms have been developed for joint code and data minimization to reduce data memory requirements over all SASs. The *dynamic programming post optimization* (DPPO) [2] performs dynamic programming over a given actor ordering (computed by any topological sort) to generate a buffer-efficient looped schedule. It has several forms for different cost functions, e.g, GDPPO [2], CDPPO [22], and SDPPO [16]. In general, construction of optimal topological sorts for SDF scheduling is NP-hard [3]. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [2] technique is an adaptable (to different cost functions), low-complexity heuristic that generates a nested looped

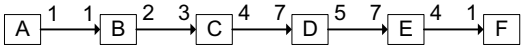



**Figure 5. A CD-DAT SDF graph.**

schedule for an acyclic graph such that precedence constraints are preserved throughout the scheduling process. In fact, DPPO and APGAN generate SASs systematically in the R-schedule form.

For a cyclic graph, a SAS may or may not exist depending on whether the numbers and locations of delays in its cycles satisfy certain sufficiency conditions. The *loose interdependence algorithm framework* (LIAF) has been developed for generating a SAS whenever one exists [2]. Beyond SASs, the work of [13] presents a recursive procedure call (RPC) based technique that generates MASs for a given R-schedule through recursive graph decomposition. The resulting procedural implementation is proven to be bounded polynomially in the graph size. This MAS technique significantly reduces memory requirement over SAS at the expense of some moderate run-time overhead. Figure 5 shows an SDF abstraction of a multi-rate CD-to-DAT application. Table 1 presents schedules computed from various SDF scheduling algorithms and their corresponding buffer requirements.

**Table 1. Schedules and buffer requirements**

| Algorithm | Schedule | Buffer |
|---|---|---|
| Flat | (147A)(147B)(98C)(56D)(40E)(160F) | 1273 |
| APGAN | (49(3AB)(2C))(8(7D)(5E(4F))) | 438 |
| DPPO | (7(7(3AB)(2C))(8D))(40E(4F)) | 347 |
| RPC-based MAS | ((2(((7((AB)(2(AB)C))D)D)(5E(4F)))(2(((7((AB)(2(AB)C))D)D)(5E(4F)))(E(4F))))(((((7((AB)(2(AB)C))D)D)(5E(4F)))(E(4F)))) | 69 |

## 3.3 Scheduling Hierarchical SDF Graphs

All of the aforesaid scheduling algorithms are designed to schedule a single, flattened (non-hierarchical) SDF graph. As explained in Section 2.1, a sophisticated DSP system is usually modeled through nested hierarchical graphs. Here, we present two approaches to schedule hierarchical SDF graphs.

The *hierarchical scheduling strategy* is developed such that the original hierarchical structure is preserved in the compiled schedule as well as in the generated code (i.e., each subhierarchy is instantiated as a subroutine). The principle of this strategy is primarily based on the SDF clustering [2]. Figure 6 presents the hierarchical scheduling strategy. The innermost *foreach* loop here is used to update the interface rates associated with each supernode. After that, the schedule $S_G$ of $G$ can be computed through any given scheduling algorithm $SAlg$. SDF clustering theory guarantees that the schedule $S$ obtained by replacing the appearance of every supernode $s$ in $S_G$ by the schedule $S_{G'}$ of the corresponding subgraph $G'$ is a valid schedule [2]. Due to the need to access every $S_G$ in the code generation phase, this algorithm collects the schedules in *scheduleMap*.

```
function hierarchicalSchedule(hierarchy topH, algorithm SAlg)
  initiate a map scheduleMap
  retrieve hierarchies from level 1 to N in topH
  for level = N to 1
    foreach hierarchy H = (G, I, M) in level
      foreach supernode s in G = (V, E)
        H' = (G', I', M') = subhrcy(s)
        compute repetition vector q_G' of G'
        foreach port p' in I'
          e = connect(p'), v' = assoc(p')
          if src(e) == s          prd(e) = prd(p') x q_G'(v')
          else if snk(e) == s     cns(e) = cns(p') x q_G'(v')
          end
        end
      end
      compute schedule S_G of G by scheduling algorithm SAlg
      put mapping H to S_G in scheduleMap
    end
  end
  return scheduleMap
end function
```

**Figure 6. The hierarchical scheduling strategy.**

In contrast, the *flattening scheduling strategy*, as presented in Figure 7, simply flattens all nested hierarchies. Then a schedule is computed for the flattened graph (without supernodes) based on any given scheduling algorithm. The DIF *hierarchy* makes it easy to perform the flattening operation by just linking inside associations of interface ports with their outside connections. For simplicity, the condition that connect($p'$) is a port is omitted in Figure 6 and Figure 7.

# 4. BUFFERING

The last step in the compilation phase is to allocate and manage buffers. Although edges in an SDF graph conceptually represent FIFO queues, implementing a FIFO structure usually leads to severe runtime and memory overhead due to maintaining the strict FIFO-based operation. In the DIF-to-C framework, only the necessary amount of memory space is allocated for each edge and buffers are managed between actor firings such that actor firings always access the correct subsets of live tokens. In this section, we describe several buffering techniques that have been implemented in the DIF-to-C framework.

```
function flatteningSchedule(hierarchy H, algorithm SAlg)
  flatten(H)
  H = (G, I, M)
  compute schedule S of G by scheduling algorithm SAlg
  return S
end function

function flatten(hierarchy topH)
  retrieve hierarchies from level 1 to N in topH
  for level = N to 1
    foreach hierarchy H = (G, I, M) in level
      foreach supernode s in G = (V, E)
        H' = (G', I', M') = subhrcy(s), G' = (V', E')
        V = V + V', E = E + E'
        foreach port p' in I'
          e = connect(p'), v' = assoc(p')
          if src(e) == s          src(e) = v', prd(e) = prd(p')
          else if snk(e) == s     snk(e) = v', cns(e) = cns(p')
          end
        end
        remove s from V
      end
    end
  end
end function
```

**Figure 7. The flattening scheduling strategy.**

## 4.1 Buffer Allocation

Various scheduling algorithms are developed for improving memory metrics based on the *non-shared memory model*, i.e., each buffer is allocated individually in memory and is live throughout a schedule. Indeed, the total buffer requirement, buffer$(G, S)$, defined in Section 3.2 is based on this non-shared memory model. Given a schedule $S$, the *non-shared buffering* technique simply allocates a buffer (declares an array) for each edge $e$ independently.

In practice, memory space can be shared by multiple buffers as long as their lifetimes do not overlapped. A *buffer sharing* technique based on lifetime analysis has been developed in [16] for delayless acyclic SDF graphs, and this technique has been shown to produce significant memory cost reductions over the non-shared memory model. In this technique, an R-schedule is first computed through SDPPO [16], and then a schedule tree is constructed to efficiently extract lifetime parameters. Next, the first-fit heuristic is applied to pack arrays efficiently into memory and determine the actual memory requirement and the buffer (array) locations. Figure 8 presents a simple example for illustrating this technique, and for a complete derivation, we refer the reader to [16].

Certain DSP computations can be executed *in-place* such that a single buffer is sufficient for both input and output. An example of this is the in-place discrete cosine transform in the Texas Instruments image processing library [25]. An *in-place* actor is naturally suitable for merging its input and output edges; this concept of merging is developed formally in [5]. Moreover, buffer merging

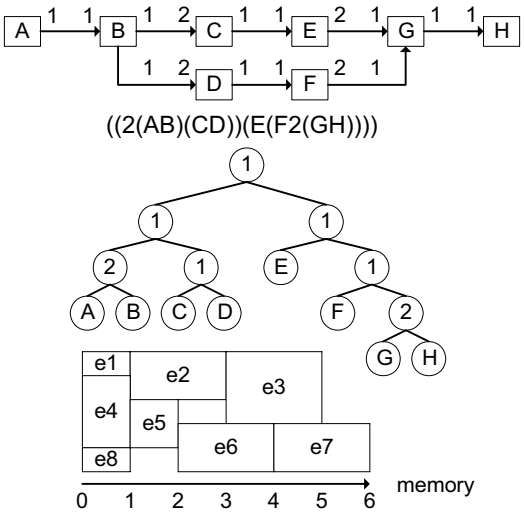



**Figure 8. A buffer sharing example.**

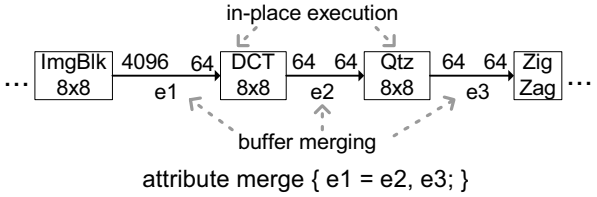



**Figure 9. An in-place buffer merging example.**

may be strictly required if the in-place actor is invoked through a predefined function that has only one argument for both input and output. A technique called *in-place buffer merging* has been developed in the DIF-to-C framework to merge buffers for a single in-place actor or a sequence of multiple in-place actors. In dataflow modeling, it is not natural to represent a single merged edge for an in-place actor, since dataflow edges are used to impose precedence constraints. Therefore, an edge attribute *merge* is dedicated in DIF to specify exactly where in-place buffer merging takes place. Figure 9 presents a sequence of in-place actors in a JPEG subsystem and the corresponding buffer merging specification in DIF.

A sequence of edges $e_1, e_2, \ldots, e_N$ in an SDF graph $G$ can be merged for in-place execution if 1. they are connected in a path, i.e., $\text{snk}(e_1) = \text{src}(e_2)$, $\text{snk}(e_2) = \text{src}(e_3)$, …, $\text{snk}(e_{N-1}) = \text{src}(e_N)$, 2. the production rate and consumption rate of each in-place actor are the same, i.e., $\text{cns}(e_1) = \text{prd}(e_2)$, $\text{cns}(e_2) = \text{prd}(e_3)$, … , $\text{cns}(e_{N-1}) = \text{prd}(e_N)$, and 3. the edges are delayless. These conditions are in general sufficient but not necessary. Once these conditions are verified, we allocate (declare) only a single buffer (array) for an edge $e_i$ and merge (assign) others to it. In our approach, $e_i$ is chosen such that the least common ancestor of $\text{src}(e_i)$ and $\text{snk}(e_i)$ is the highest internal node in the schedule tree. Because of condition 2 and the properties of R-schedules, $e_i$ is guaranteed to have the maximum $\text{maxToken}(e_i, S)$ among $e_1, e_2, \ldots, e_N$.

## 4.2 Buffer Management

Knowledge of just the buffer size and the buffer (array) address (denoted as buffer$(e)$) is not enough for $\text{src}(e)$ and $\text{snk}(e)$ to access the right place in the buffer at a particular iteration. Buffer management through circular buffering has been developed and [4] presents in-depth discussions of this and other forms of SDF buffering. Figure 10 illustrates circular buffering under several cases.

In the DIF-to-C framework, actors are associated with C functions, and inputs and outputs of actors are passed by pointer through function arguments. This is a widely used convention in implementing DSP library modules, e.g., see [24,25]. This convention generally assumes that input/output data are consecutive in memory space. However, this assumption prevents us from directly applying the circular buffering approach, since a particular firing may access tokens that wrap around the buffer, e.g., $V_2$ in Figure

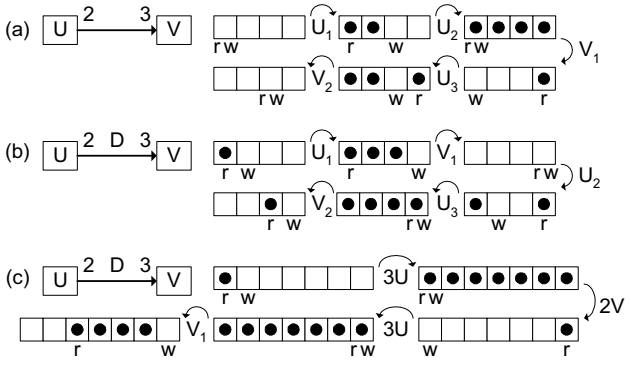



**Figure 10. Circular buffering**
**(a) MAS, (b) MAS+delay, (c) SAS+delay**

10(a), $U_2$ in Figure 10(b), and $V_1$ in Figure 10(c).

We have developed a modified memory management approach such that circular buffering is preserved and input/output data are still consumed/produced consecutively. In the initial stage, a buffer (array) is allocated (declared) for an edge $e$ with enlarged size, $\text{maxToken}(e) + \max\{\text{prd}(e), \text{cns}(e)\} - 1$, to accommodate wrapped-around tokens for the worst case situation. The read and write pointers, denoted as $\text{rp}(e)$ and $\text{wp}(e)$ respectively, are initialized as: $\text{rp}(e) = 0$ and

$$\text{wp}(e) = \begin{cases} 0, \text{ if } \text{delay}(e) = 0 \text{ or } \text{delay}(e) = \text{maxToken}(e) \\ \text{delay}(e), \text{ if } 0 < \text{delay}(e) < \text{maxToken}(e, S) \end{cases}.$$

For each firing of $\text{src}(e)$, it is directed to write the buffer at $\text{buffer}(e) + \text{wp}(e)$, and for each firing of $\text{snk}(e)$, it is directed to read the buffer at $\text{buffer}(e) + \text{rp}(e)$. Before a firing of $\text{snk}(e)$, if $\text{rp}(e) + \text{cns}(e) > \text{maxToken}(e, S)$, the first $\text{rp}(e) + \text{cns}(e) - \text{maxToken}(e, S)$ tokens are copied to the position after $\text{maxToken}(e, S) - 1$ for wrap-around access. Similarly, after a firing of $\text{src}(e)$, if $\text{wp}(e) + \text{prd}(e) > \text{maxToken}(e, S)$, $\text{wp}(e) + \text{prd}(e) - \text{maxToken}(e, S)$ tokens after the position $\text{maxToken}(e, S) - 1$ are copied to the front. In addition, $\text{rp}(e)$ and $\text{wp}(e)$ are updated as $\text{rp}(e) = (\text{rp}(e) + \text{cns}(e)) \bmod \text{maxToken}(e, S)$ and $\text{wp}(e) = (\text{wp}(e) + \text{prd}(e)) \bmod \text{maxToken}(e, S)$.

Through experiments, we have demonstrated that this approach is capable of supporting various software synthesis scenarios, especially for MAS and the presence of arbitrary delays. However, this approach also introduces buffer overhead for accommodating consecutive access and runtime overhead due to modulo and memory copy operations.

If the input graph is delayless and the given schedule is a SAS, we have that $\text{maxToken}(e, S)$ is sufficient for periodic firings without wrap-around access, and read and write pointers can be statically reset without performing modulo operations. Since a broad range of DSP applications are modeled as acyclic, delayless SDF graphs and SASs are usually preferable, this *static read/write pointer resetting* technique has been implemented in the DIF-to-C framework for improving runtime and memory performance. Given a delayless graph $G$ and a SAS $S$, an edge $e$ is only live in the subschedule $L$ that corresponds to the least common ancestor of $\text{src}(e)$ and $\text{snk}(e)$ in the schedule tree, because neither $\text{src}(e)$ nor $\text{snk}(e)$ appears beyond $L$ in $S$. In addition, $\text{maxToken}(e, S)$ is equal to the total number of tokens exchanged between $\text{src}(e)$ and $\text{snk}(e)$ within $L$. Based on these facts, we allocate $\text{maxToken}(e, S)$ for $e$, reset $\text{rp}(e)$ and $\text{wp}(e)$ at the beginning of the loop $L$, and update them after each firing of $\text{src}(e)$ and $\text{snk}(e)$ without any modulo operation, i.e., $\text{rp}(e) = \text{rp}(e) + \text{cns}(e)$ and $\text{wp}(e) = \text{wp}(e) + \text{prd}(e)$, respectively.

# 5. CODE GENERATION

Code generation is the final phase in the DIF-to-C software synthesis framework. By integrating the DIF representations, scheduling algorithms, and buffering techniques, the code generator is able to generate a C-code implementation of the coarse-grain dataflow graph of a DSP system design. The generated code is mainly a looped sequence of function invocations (determined by the schedule) interleaved with buffer management routines. Finally, an executable is compiled from the generated code together with fine-grain actors (functions) or library links. In this section, we describe our code generation algorithm in detail and introduce how several strategies in this regard are developed in a systematic way.

## 5.1 Function Prototypes and Datatypes

Unlike general design tools that only provide their own actor libraries, the DIF-to-C software synthesis framework is designed to support most C-based DSP libraries. In order to accommodate various C-based actors (functions), we impose the least possible constraints on function prototypes and function designs.

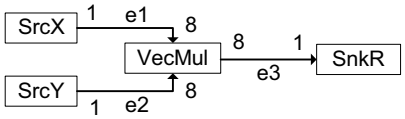
For functions operating on primitive C datatypes (e.g., int, float, etc.) or typedef structures, the only two constraints are 1. input and output tokens should be passed by pointer through function arguments, and for SDF-based synthesis; and 2. the production and consumption rates should be known at compile time. In general, most C-based functions naturally conform these constraints. Figure 11 illustrates the prototype of the vector multiplication function in the Texas Instruments DSP Library [24]. The inputs $x$ and $y$, and output $r$ are passed by pointer through a "*float**". The argument $n$ indicates the number of elements in $x$, $y$, and $r$, which also implies that the production/consumption rate of $x$, $y$, and $r$ is $n$. Figure 11 also shows an SDF example and the corresponding actor specification. Note that the order of actor attributes should preserve the order of arguments in the function prototype.

When discussing buffer size in the previous sections, we only take the number of tokens into account. In practice, the data type is also necessary in code generation. The attribute *datatype* is dedicated in DIF for specifying the data type of an edge $e$, denoted as $\text{type}(e)$, and the datatype of a port $p$, denoted as $\text{type}(p)$, as shown in Figure 4. In code generation, a buffer for $e$ is declared as "$\text{type}(e)$ $e$[size]", where the buffer size is determined based on Section 4; when instantiating a subroutine for a subhierarchy, "$\text{type}(p)$ $*p$" is generated as a subroutine argument for passing the buffer pointer of the outside connection.

## 5.2 DIFtoC Code Generator

In the DIF-to-C software synthesis framework, *DIFtoC* is the base class for performing code generation. It is developed based on the hierarchical scheduling strategy, non-shared buffer allocation, and

```
void DSPF_sp_vecmul(float *x, float *y, float *r, int n)
```




```
actor VecMul {
  computation = "DSPF_sp_vecmul";
  x = e1;
  y = e2;
  r = e3;
  n = 8;
}
```

**Figure 11. Function Prototype and actor specification.**

```
function generate(hierarchy topH, algorithm SAlg)
  if flatteningSchedulingStrategy  flatten(topH)  end
  scheduleMap = hierarchicalSchedule(topH, SAlg)
  generateMain(topH, scheduleMap.get(topH))
  retrieve hierarchies from level 2 to N in topH
  for level = N to 2
    foreach hierarchy H = (G, I, M) in level
      generateSubRoutine(H, scheduleMap.get(H))
    end
  end
end function

function generateMain(hierarchy H, schedule S)
  H = (G, I, M)
  allocateBuffer(G)
  generate "for loop #iteration"
  generateSchedule(H, S)
end function

function generateSubRoutine(hierarchy H, schedule S)
  H = (G, I, M)
  generate subroutine prototype for all interface ports in I in order:
      "H(type(p1) *p1, type(p2) *p2, ..., type(pN) *pN)"
  foreach port pi in I
    if pi is an input port          declare "rp(pi) = 0"
    elseif pi is an output port     declare "wp(pi) = 0"
    end
  end
  allocateBuffer(G)
  generateSchedule(H, S)
end function

function generateSchedule(hierarchy H, schedule S)
  S = nT1T2...TN
  generate "for loop n"
  foreach schedule element Ti in S
    if Ti is a firing               generateFiring(H, Ti)
    elseif Ti is a schedule         generateSchedule(H, Ti)
    end
  end
end function

function generateFiring(hierarchy H, firing F)
  F = nv
  generate "for loop n"
  if v is a supernode       generateSubRoutineCall(v)
  else                      generateFunctionCall(v)
  end
end function

function allocateBuffer(graph G)
  G = (V, E)
  foreach edge e in E
    allocate buffer:
      "type(e) e [maxToken(e, S) + max(prd(e), cns(e)) -1)]"
    declare "rp(e) = 0"
    if delay(e) == 0 or delay(e) == maxToken(e, S)
      declare "wp(e) = 0"
    else      declare "wp(e) = delay(e)"
    end
  end
end function
```

**Figure 12. DIFtoC pseudocode - 1.**

```
function generateFunctionCall(node v)
  (computation, att1, att2, ..., attN ) = attributes(v)
  generate "computation"                  //function name
  foreach atti                            //function parameters
    if atti is an edge e and snk(e) == v
      generate "e+rp(e)", manageBuffer(e, v)
    elseif atti is an edge e and src(e) == v
      generate "e+wp(e)", manageBuffer(e, v)
    elseif atti is an input port p and assoc(p) == v
      generate "p+rp(p)", manageBuffer(p, v)
    elseif atti is an output port p and assoc(p) == v
      generate "p+wp(p)", manageBuffer(p, v)
    elseif atti is a parameter param      generate "param"
    end
  end
end function

function generateSubRoutineCall(supernode s)
  H' = (G', I', M') = subhrcy(s)
  generate "H'"                           //subroutine name
  foreach p' in I' in order               //subroutine parameters
    if p' is an input port and connect(p') is an edge e
      generate "e+rp(e)", manageBuffer(e, s)
    elseif p' is an output port and connect(p') is an edge e
      generate "e+wp(e)", manageBuffer(e, s)
    elseif p' is an input port and connect(p') is a port pi
      generate "pi+rp(pi)", manageBuffer(pi, s)
    elseif p' is an output port and connect(p') is a port po
      generate "po+wp(po)", manageBuffer(po, s)
    end
  end
end function

function manageBuffer(edge e, node v)
  if snk(e) == v
    generate:
      "if (rp(e)+cns(e)>maxToken(e,S)) {
      for (j=0; j<rp(e)+cns(e)-maxToken(e,S); j++) {
      e[j+maxToken(e,S)] = e[j]; } }"
    before function/subroutine call
    generate:
      "rp(e) = (rp(e)+cns(e)) mod maxToken(e,S)"
    after function/subroutine call
  elseif src(e) == v
    generate:
      "if (wp(e)+prd(e)>maxToken(e,S)) {
      for (j=0; j<wp(e)+prd(e)-maxToken(e,S); j++) {
      e[j] = e[j+maxToken(e,S)]; } } "
      "wp(e) = (wp(e)+prd(e)) mod maxToken(e,S)"
    after function/subroutine call
  end
end function

function manageBuffer(port p, node v)
  if p is an input port          generate:
    "rp(p) = rp(p) + cns(p)"     after function/subroutine call
  elseif p is an output port     generate:
    "wp(p) = wp(p) + prd(p)"     after function/subroutine call
  end
end function
```

**Figure 13. DIFtoC pseudocode - 2.**

the modified circular buffer management such that it is capable of supporting any SAS or MAS, any configuration of edge delays, and both flat and hierarchical SDF graphs. Figure 12 and Figure 13 present the code generation algorithm that underlies the *DIFtoC* code generator. Briefly speaking, a *main()* function is generated for the top-level hierarchy *topH*, and a subroutine is constructed for each subhierarchy. For each loop in the schedule, a loop construct is instantiated, and for each actor in the schedule, a function call or a subroutine call is instantiated. Buffers are declared as arrays, and code for managing circular buffers and updating read and write pointers is generated between function/subroutine invocations. Note that the flattening scheduling strategy is also supported by flattening the top level hierarchy in the early stages of the function *generate* in the *DIFtoC*.

The *DIFtoC* schedules dataflow graphs based on the user-specified scheduling algorithm *SAlg*, and thus provides flexibility in terms of scheduling. Moreover, integrating different combinations of buffering strategies can further broaden the possible design space. In our framework, such developments can be implemented naturally by extending and overriding *DIFtoC*. Figure 14 presents the classes in the current DIF-to-C software synthesis framework. Regarding SASs (in R-schedule form) and delayless graphs, the following classes have been developed for better memory and runtime performance. *DIFtoCsrw* extends *DIFtoC* and implements static read/write pointer resetting as described in Section 4.2 by simply overriding functions *allocateBuffer*, *manageBuffer*, and *generateSchedule*. *DIFtoCbs* extends *DIFtoCsrw* and overrides function *allocateBuffer* to implement the buffer sharing technique [16]. *DIFtoCipbm* also extends *DIFtoCsrw* and implements the in-place buffer merging technique as explained in Section 4.1 for synthesizing graphs containing in-place execution actors.

# 6. EXPERIMENT RESULTS

In this section, we demonstrate our DIF-to-C software synthesis framework by synthesizing real-world DSP applications and then compiling and simulating the generated C codes. The DSP applications we have experimented with include the highly multi-rate CD-DAT and DAT-CD sample rate conversion systems, a four-level tree-structured filter bank performing biorthogonal wavelet decomposition, a synthetic aperture radar (SAR) system containing range and azimuth processing, and a JPEG encoder subsystem consisting of RGB-YCbCr, 2D-DCT, Quantization, and ZigZag sequencing. We program the coarse-grain SDF graphs of these applications in DIF, and then generate various C implementations based on different combinations of scheduling and buffering strategies through the DIF-to-C framework. Together with fine-grain actors either obtained from Texas Instruments DSP and image processing libraries [24,25] or manually implemented in C, we compile and simulate them in the Texas Instruments Code Composer Studio. The target simulation platform used in our experiments is the TMS320C64x DSP series platform and the compiler optimization setting is *none*.

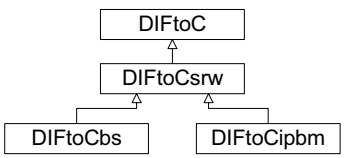



**Figure 14. The class hierarchy in the DIF-to-C framework.**

Figure 15 presents the SDF graphs and the simulation results of the (a) CD-DAT, (b) DAT-CD, (c) filter bank, (d) SAR, and (e) JPEG applications. In our experiment, the combinations of scheduling and buffering strategies include DIFtoC (modified circular buffering) with the flat scheduling strategy (abbreviated as C-F in Figure 15), DIFtoC with APGAN plus DPPO post-optimization (abbreviated as C-AD), DIFtoC with RPC-based MAS (abbreviated as C-RPC), DIFtoCsrw (static read/write pointer resetting) with APGAN-DPPO (abbreviated as SRW-AD), DIFtoCbs (buffer sharing) with SDPPO (abbreviated as BS-SD), and DIFtoCipbm (in-place buffer merging for using in-place actors in the JPEG application) with APGAN-DPPO (abbreviated as IPBM-AD). Since the filter bank and SAR systems are modeled using hierarchical SDF graphs, we also present both the hierarchical and flattening scheduling results (abbreviated as Hierarchical and Flattened respectively in Figure15). Note that the actual possible combinations are much more than what is shown in Figure 15.

The metrics we examined are memory (in bytes), code size (in bytes), and CPU cycles. The memory metric represents the total memory space allocated for all dataflow edges. The code size metric represents the compilation size of the C-code generated by the DIF-to-C software synthesis framework. In other words, it includes all of the automatically generated main() function and subroutines, but excludes fine-grain functions unrelated to the DIF-to-C framework (i.e., those that are either obtained from libraries or implemented by hand). The CPU metric measures the cycles required for one iteration of a periodic schedule of the application dataflow graph. It includes the CPU cycles spent only in the generated code (abbreviated as *CPU-excluded*) and the total CPU cycles for the complete executable (abbreviated as *CPU-total*). Note that the CPU-excluded component reflects just the inter-actor dataflow graph functionality without fine-grain actor execution.

According to Figure 15, we found that there exists a complex range of trade-offs and generally no particular technique dominates for all applications. For the CD-DAT and DAT-CD systems, RPC-based MAS significantly reduces memory requirements at the expense of code size. For the filter bank application, the buffer sharing method is an efficient approach, and the flattening strategy generally performs better than the hierarchical strategy. Even though the *DIFtoC* code generator allows MAS, it causes severe overhead in the SAR and JPEG applications. In these two cases, static read/write pointer resetting and buffer sharing can improve the situation significantly. For the JPEG application, since several operations can be executed in-place, the buffer-merging technique is very suitable. Regarding to the CPU-total metric for all applications, we found that the dataflow overhead (schedules, buffer allocation, and buffer management) is insignificant when taking large repetitions of heavily-computational actors into account. In general, such heavily computation-involved actors are usually optimized through compiler techniques or by hand.

# 7. CONCLUSION

In this paper, we have reviewed the principles behind the dataflow interchange format (DIF) and the DIF-to-C software synthesis framework. We have then described the programming, compila-

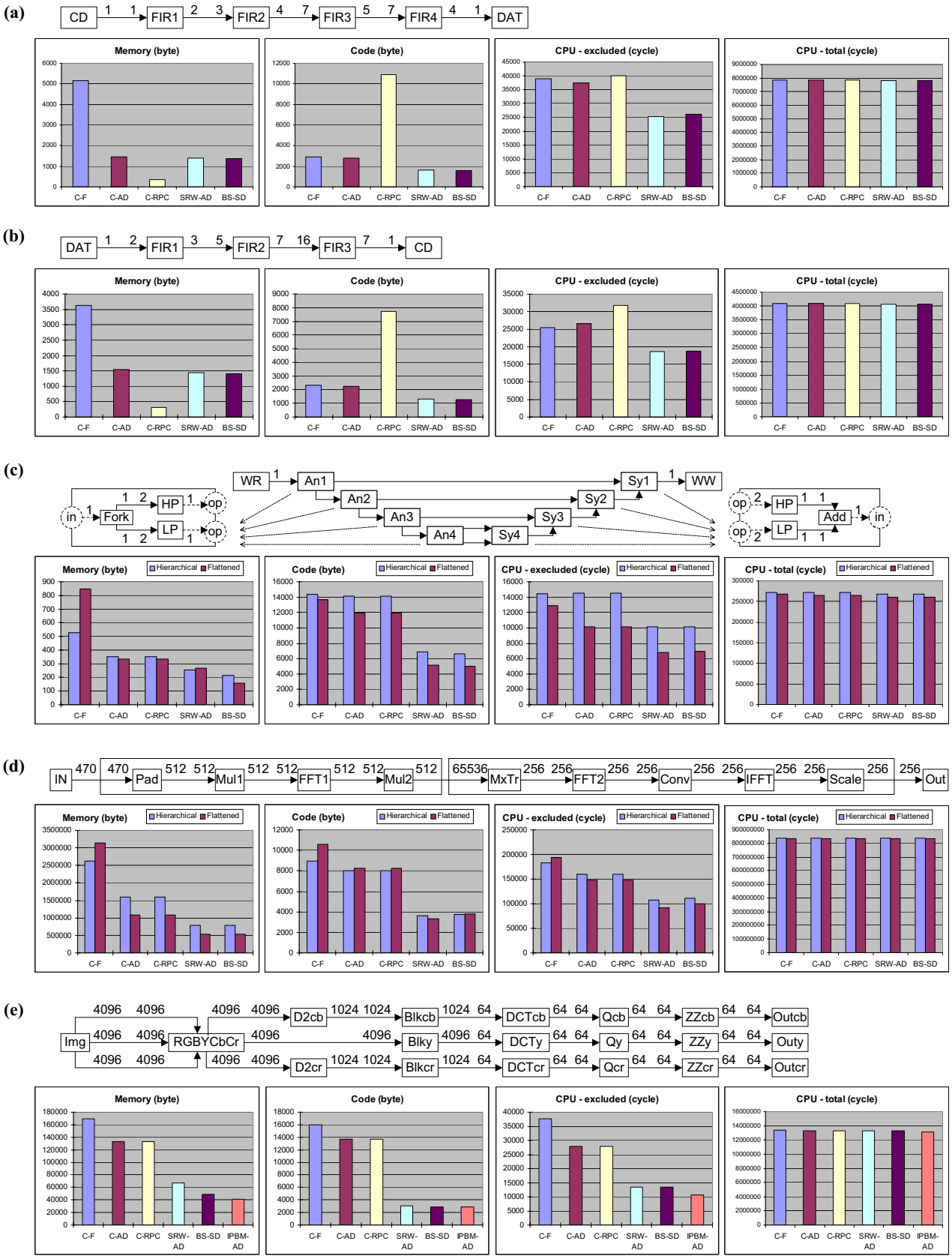



**Figure 15. Simulation results. (a) CD-DAT, (b) DAT-CD, (c) filter bank, (d) SAR, (e) JPEG.**

tion, and code generation phases of the software synthesis framework. Finally, we have demonstrated the synthesis automation and the broad range of scheduling and buffering trade-offs offered by our DIF-to-C framework. Our ongoing work includes the development of a DIFtoVSIPL code generator for synthesis from DIF specifications to VSIPL [12] implementations, and research in synthesis techniques for dataflow models of computation that have increased expressive power compared to SDF.

DIF is being developed in the University of Maryland DSP-CAD Research Group. Currently, DIF is being evaluated and used by a number of research partners. A general public release of the DIF package is being planned for the near future.

## 8. ACKNOWLEDGEMENTS

This research was supported in part by the U. S. Defense Advanced Research Projects Agency (DARPA) via the U. S. Army Aviation and Missile Command (Contract Number DAAH01-03-C-R236).

## 9. REFERENCES

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

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

[3] Bhattacharyya, S. S., Leupers, R., and Marwedel, P. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.

[4] Bhattacharyya, S. S. and Lee, E. A. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, 42(5):1190-1201, May 1994.

[5] Bhattacharyya, S. S. and Murthy, P. K. The CBP parameter — a module characterization approach for DSP software optimization. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 38(2):131-146, September 2004.

[6] Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. A. Cyclo-static data flow. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 3255-3258, May 1995.

[7] Buck, J. and Vaidyanathan, R. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, May 2000.

[8] Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. Taming heterogeneity — the Ptolemy approach. *Proceedings of the IEEE*, v.91, No. 2, January 2003.

[9] Hsu, C. and Bhattacharyya, S. S. Porting DSP applications across design tools using the Dataflow Interchange Format. In *Proceedings of the International Workshop on Rapid System Prototyping*, Montreal, Canada, June 2005.

[10] Hsu, C. and Bhattacharyya, S. S. *Dataflow Interchange Format Version 0.2*. Technical Report, UMIACS-TR-2004-66, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2004.

[11] Hsu, C., Keceli, F., Ko, M., Shahparnia, S., and Bhattacharyya, S. S. DIF: An interchange format for dataflow-based design tools. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 423-432, Samos, Greece, July 2004.

[12] Janka, R., Judd, R., Lebak, J., Richards, M., and Campbell, D. VSIPL: An object-based open standard API for vector, signal, and image processing. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 949–952.

[13] Ko, M., Murthy, P. K., and Bhattacharyya, S. S. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Processors*, pages 47-61, Amsterdam, The Netherlands, September 2004.

[14] Lauwereins, R., Engels, M., Ade, M., and Peperstraete, J. A. Grape-II: A system-level prototyping environment for DSP applications. *IEEE Computer Magazine*, 28(2):35-43, February 1995.

[15] Lee, E. A., and Messerschmitt, D. G. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.

[16] Murthy, P. K., and Bhattacharyya, S. S. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177-198, February 2001.

[17] Murthy, P. K. and Lee, E. A. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064-2079, August 2002.

[18] Robbins, C. B. *Autocoding Toolset Software Tools for Automatic Generation of Parallel Application Software*. Technical report, Management Communications and Control, Inc., 2002.

[19] Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., and Deprettere, E. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference*, February 2004.

[20] Sung, W., Oh, M., Im, C., and Ha, S. Demonstration of hardware software codesign workflow in PeaCE. In *Proceedings of International Conference on VLSI and CAD*, October 1997.

[21] Thies, W., Karczmarek, M., and Amarasinghe, S. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, April 2002.

[22] Zitzler, E., Teich, J., and Bhattacharyya, S. S. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, pages 83-98, February 2000.

[23] Agilent Technologies. *ADS Ptolemy Simulation*. September 2004.

[24] Texas Instruments. *TMS320C67x DSP Library Programmer's Reference Guide*. February 2003.

[25] Texas Instruments. *TMS320C64x Image/Video Processing Library Programmer's Reference*. October 2003.