In Y. H. Hu, editor, Programmable Digital Signal Processors:
Architecture, Programming, and Applications, pages 333-378.
Marcel Dekker, Inc., 2002.

Hardware/Software Co-synthesis of DSP Systems

Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park

This chapter focuses on the automated mapping of high level specifications of DSP applications into implementation platforms that employ programmable DSPs. Since programmable DSPs are often used in conjunction with other types of programmable processors, such as microcontrollers and general-purpose microprocessors, and with various types of hardware modules, such as FPGAs, and ASIC circuitry, this mapping task, in general, is one of *cosynthesis* — the joint synthesis of both hardware and software — for a heterogeneous multiprocessor.

Since a large variety of cosynthesis techniques have been developed to date, it is not possible here to provide comprehensive coverage of the field. Instead, we focus on a subset of topics that are central to DSP-oriented cosynthesis — application modeling, hardware/software partitioning, synchronization optimization, and block-processing. Some important topics related to cosynthesis that are not covered here include memory management [1, 11, 26, 37, 53], which is discussed in Chapter 10; DSP code generation from procedural language specifications [39], which is the topic of Chapter 7; and performance analysis [36, 49, 54].

Additionally, we focus on synthesis from *coarse-grain dataflow models* due to the increasing importance of such modeling in DSP design tools, and the ability of such modeling to expose valuable, high-level structure of DSP applications that is difficult to deduce from within compilers for general purpose programming models, and other types of models. Thus, we do not explore techniques for fine-grain cosynthesis [21], including synthesis of application-specific instruction processors (ASIPs) [43], nor do we explore cosynthesis for control-dominant systems, such as those based on procedural language specifications [22], communicating sequential processes [50], and finite state machine models [6]. All of these are important directions within cosynthesis research, but they do not fit centrally within the DSP-oriented scope of this chapter.

Motivation for coarse-grain dataflow specification stems from the growing trend towards specifying, analyzing, and verifying embedded system designs in terms of domain-specific concurrency models [33], and the increasing use of

dataflow-based concurrency models in high-level design environments for DSP system implementation. Such design environments, which enable DSP systems to be specified as hierarchies of block diagrams, offer several important advantages, including intuitive appeal, and natural support for desirable software engineering practices such as library-based design, modularity, and design reuse

Potentially, the most useful benefit of dataflow-based graphical programming environments for DSP is that carefully-specified graphical programs can expose coarse-grain structure of the underlying algorithm, and this structure can be exploited to facilitate synthesis and formal verification in a wide variety of ways. For example, the cosynthesis tasks of *partitioning* and *scheduling* — determining the resources on which the computations in an application will execute, and the execution ordering of computations assigned to the same resource — typically have a large impact on all of the key implementation metrics of a DSP system. A dataflow-based system specification exposes high-level partitioning and scheduling flexibility that is often not possible to deduce manually or automatically from procedural language (e.g., assembly language or C) specifications. This flexibility can be exploited by cosynthesis tools to streamline an implementation based on the given set of performance and cost objectives. We will elaborate on partitioning and scheduling of dataflow-based specifications in Sections 3, 4, and 6.

The organization of the remainder of this chapter is as follows. We begin with a brief summary of our notation in working with fundamental, discrete math concepts. Then we discuss the principles of coarse-grain dataflow modeling that underlie many high-level DSP design tools. This discussion includes a detailed treatment of synchronous dataflow and cyclo-static dataflow, which are two of the most popular forms of dataflow employed in DSP design. Next, we review three techniques — GCLP, COSYN, and the evolutionary algorithm approach of CodeSign — for automated partitioning of coarse-grain dataflow specifications into hardware and software. In Section 5, we present an overview of techniques for efficiently synchronizing multiple processing elements in heterogeneous multiprocessor systems, such as those that result from hardware/software cosynthesis, and in Section 6, we discuss techniques for optimizing the application of *block processing*, which is a key opportunity for improving the throughput of cosynthesis solutions. Finally, we conclude in Section 7 with a summary of the main developments in the chapter. Throughout the chapter, we occasionally incorporate minor semantic modifications of the techniques that we discuss — without changing their essential behavior — to promote conciseness, clarity, and more uniform notation.

1 Background

We denote the set of non-negative integers $\{0, 1, 2, \ldots\}$ by the symbol $\aleph$, the set of extended non-negative integers $(\aleph \cup \{\infty\})$ by $\overline{\aleph}$, the set of positive integers by Z^+, the set of extended integers $(\{-\infty, \infty\} \cup \{\ldots, -1, 0, 1, \ldots\})$ by $\overline{Z}$, and the cardinality of (number of elements in) a finite set S by $|S|$. By a *directed graph*, we mean an ordered pair (V, E), where V is a set of objects called *vertices*, and E is a set of ordered pairs, called *edges*, of elements in V. We use the usual pictorial representation of directed graphs in which circles represent vertices, and arrows represent edges. For example, Figure 1 represents a directed graph with vertex set $V = \{a, b, c, d, e, f, g, h\}$, and edge set

$$E = \{(c, a), (b, c), (a, b), (b, h), (d, f), (f, e), (e, f), (e, d)\}. \quad (1)$$

If $e = (v_1, v_2)$ is an edge in a directed graph, we write $src(e) = v_1$, and $snk(e) = v_2$; and we say that $src(e)$ is the *source* vertex of e, $snk(e)$ is the *sink* vertex of e; e is *directed from* $src(e)$ *to* $snk(e)$; e is an *outgoing edge* of $src(e)$; and e is an *incoming edge* of $snk(e)$.

Given a directed graph $G = (V, E)$, and a vertex $v \in V$, we define the *incoming* and *outgoing edge sets* of v by

$$in(v) = \{e \in E | snk(e) = v\}, \text{ and } out(v) = \{e \in E | src(e) = v\}, \quad (2)$$

respectively. Furthermore, given two vertices v_1 and v_2 in G, we say that v_1 is a *predecessor* of v_2 if there exists $e \in E$ such that $src(e) = v_1$ and $snk(e) = v_2$; we say that v_1 is a *successor* of v_2 if v_2 is a predecessor of v_1; and we say that v_1 and v_2 are *adjacent* if v_1 is a successor or predecessor of v_2. A *path* in (V,E) is a finite sequence $(e_1, e_2, \ldots, e_n) \in E$ such that

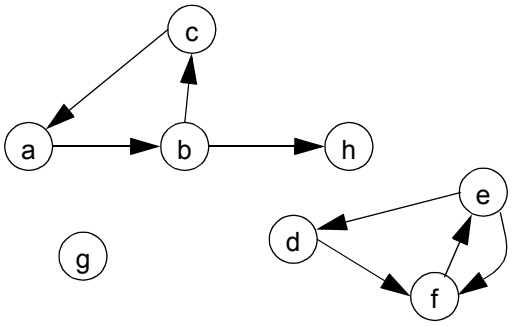

Figure 1. An example of a directed graph.

$$\text{for } i = 1, 2, \ldots, (n-1),\ snk(e_i) = src(e_{i+1}). \tag{3}$$

Thus, $((a, b))$, $((d, f), (f, e), (e, f), (f, e))$, $((b, c), (c, a), (a, b))$, and $((a, b), (b, h))$ are examples of paths in Figure 1.

We say that a path $p = (e_1, e_2, \ldots, e_n)$ *originates at* the vertex $src(e_1)$, and *terminates at* $snk(e_n)$, and we write

$$edges(p) = \{e_1, e_2, \ldots, e_n\}, \text{ and}$$
$$vertices(p) = \{src(e_1), src(e_2), \ldots, src(e_n), snk(e_n)\}. \tag{4}$$

A *cycle* is a path that originates and terminates at the same vertex. A cycle $(e_1, e_2, \ldots, e_n)$ is a *simple cycle* if $src(e_i) \neq src(e_j)$ for all $i \neq j$. In Figure 1, $((c, a), (a, b), (b, c))$, $((a, b), (b, c), (c, a))$, and $((f, e), (e, f))$ are examples of simple cycles. The path $((d, f), (f, e), (e, f), (f, e), (e, d))$ is a cycle that is not a simple cycle.

By a *subgraph* of a directed graph $G = (V, E)$, we mean the directed graph formed by any subset $V' \subseteq V$ together with the set of edges $\{e \in E | (src(e), snk(e) \in V')\}$. For example, the directed graph

$$(\{e, f\}, \{(e, f), (f, e)\}) \tag{5}$$

is a subgraph of the directed graph shown in Figure 1.

Given a directed graph $G = (V, E)$, a sequence of vertices $(v_1, v_2 \ldots, v_k)$ is a *chain* that joins v_1 and v_k if v_{i+1} is adjacent to v_i for $i = 1, 2, \ldots, (k-1)$. We say that a directed graph is *connected* if for any pair of distinct members A, B of V, there is a chain that joins A and B. Thus, the directed graph in Figure 1 is not connected (e.g., since there is no chain that joins g and b), while the subgraph associated with the vertex subset $\{a, b, c, h\}$ is connected.

A *strongly connected* directed graph C has the property that between every distinct pair of vertices w and v in C, there is a directed path from w to v and a directed path from v to w. A *strongly connected component* **(***SCC***)** of a directed graph is a maximal strongly connected subgraph. The directed graph in Figure 1 contains four SCCs. Two of these SCCS — $(\{g\}, \varnothing)$ and $(\{h\}, \varnothing)$ — are called *trivial* SCCs since each contains a single vertex and no edges. The other two SCCS in Figure 1 are the directed graphs (V_1, E_1) and (V_2, E_2), where $V_1 = \{a, b, c\}$, $E_1 = \{(a, b), (b, c), (c, a)\}$, $V_2 = \{d, e, f\}$, and $E_2 = \{(e, f), (f, e), (e, d), (d, f)\}$.

Many excellent textbooks, such as [17, 52], provide elaboration on the graph-theoretic fundamentals summarized in this section.

2 Coarse-grain dataflow modeling for DSP

2.1 Dataflow modeling principles

In the dataflow paradigm, a computational specification is represented as a directed graph. Vertices in the graph (called *actors*) correspond to computational modules in the specification. In most dataflow-based DSP design environments, actors can be of arbitrary complexity. Typically, they range from elementary operations such as addition or multiplication to DSP subsystems such as FFT units or adaptive filters.

An edge (v_1, v_2) in a dataflow graph represents the communication of data from v_1 to v_2. More specifically, an edge represents a FIFO (first-in-first-out) queue that buffers data values (*tokens*) as they pass from the output of one actor to the input of another. When dataflow graphs are used to represent signal processing applications, a dataflow edge e has a non-negative integer delay $del(e)$ associated with it. The delay of an edge gives the number of initial data values that are queued on the edge. Each unit of dataflow delay is functionally equivalent to the z^{-1} operator in DSP: the sequence of data values $\{y_n\}$ generated at the input of the actor $snk(e)$ is equal to the shifted sequence $\{x_{n-del(e)}\}$, where $\{x_n\}$ is the data sequence generated at the output of the actor $src(e)$.

A dataflow actor is *enabled* for execution any time it has sufficient data on its incoming edges (i.e., in the associated FIFO queues) to perform its specified computation. An actor can execute (*fire*) at any time when it is enabled (*data-driven execution*). In general, the execution of an actor results in some number of tokens being removed (*consumed*) from each incoming edge, and some number being placed (*produced*) on each outgoing edge. This production activity in general leads to the enabling of other actors.

The order in which actors execute is not part of a dataflow specification, and is constrained only by the simple principle of data-driven execution defined above. This is in contrast to many alternative programming models, such those that underlie procedural languages, in which execution order is *overspecified* by the programmer [4]. The actor execution order for a dataflow specification may be determined at compile time (if sufficient static information is available), at run-time, or using a mixture of compile-time and run-time techniques.

2.2 Synchronous dataflow

Synchronous dataflow (*SDF*), introduced by Lee and Messerschmitt [34], is the simplest, and currently, the most popular form of dataflow modeling for DSP design. SDF imposes the restriction that the number of data values produced by an actor onto each outgoing edge is constant, and similarly, the number of data values consumed by an actor from each incoming edge is constant. Thus, an SDF

edge e has two additional attributes — the number of data values produced onto e by each firing of the source actor, denoted $prd(e)$, and the number of data values consumed from e by each firing of the sink actor, denoted $cns(e)$.

Example 1: A simple example of an SDF abstraction is shown in Figure 2. Here, each edge is annotated with the number of data values produced and consumed by the source and sink actors, respectively. For example, $prd((B, C)) = 1$, and $cns((B, C)) = 2$. The "2D" next to the edge (D, E) represents two units of delay. Thus, $del((D, E)) = 2$.

The restrictions imposed by the SDF model offer a number of important advantages, including *static scheduling*, which avoids the execution time and power consumption overhead, and the unpredictability of dynamic scheduling approaches; and decidability of key verification problems — in particular, determination of bounded memory requirements and deadlock avoidance. These two verification problems are critical in the development of DSP applications since DSP systems involve iterative operation on vast, often unbounded, sequences of input data. Not all SDF graphs permit *admissible* operation on unbounded input sets — that is, operation without deadlock, and without unbounded data accumulation on one or more edges. However, it can always be determined at compile time whether or not admissible operation is possible for a given SDF graph. In exchange for its strong advantages, the SDF model has limited expressive power — not all applications can be expressed in the model.

A necessary and sufficient condition for admissible operation to be possible for an SDF graph is the existence of a *valid schedule* for the graph, which is a

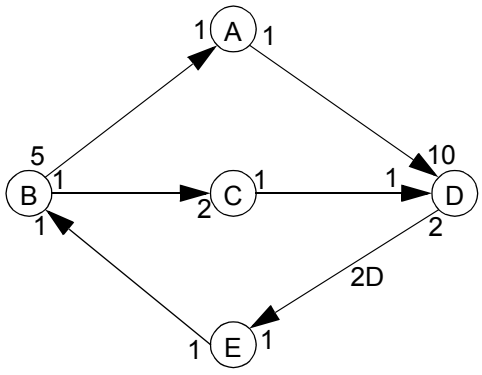

Figure 2. An example of an SDF graph.

finite sequence of actor firings that executes each actor at least once, fires actors only after they are enabled, and produces no net change in the number of tokens queued on each edge. SDF graphs for which valid schedules exist are called *consistent* SDF graphs.

Efficient algorithms have been developed by Lee and Messerschmitt [34] to determine whether or not a given SDF graph is consistent, and to determine the minimum number of times that each actor must be fired in a valid schedule. We represent these minimum numbers of firings by a vector (called the *repetitions vector*) $\mathbf{q}_G$, indexed by the actors in G (we often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the *balance equations* for G, which specify that $\mathbf{q}$ must satisfy

$$\mathbf{q}(src(e)) \times prd(e) = \mathbf{q}(snk(e)) \times cns(e), \text{ for every edge } e \text{ in } G. \quad (6)$$

Associated with any valid schedule S, there is a positive integer $J(S)$ such that S fires each actor A exactly $(J(S) \times \mathbf{q}(A))$ times. This number $J(S)$ is referred to as the *blocking factor* of S.

Given a consistent SDF graph G, the *total number of samples exchanged* (per schedule iteration) on an SDF edge e in G, denoted $TNSE_G(e)$, is defined by the equal-valued products in the LHS and RHS of (6). That is,

$$TNSE_G(e) = \mathbf{q}(src(e)) \times prd(e) = \mathbf{q}(snk(e)) \times cns(e). \quad (7)$$

Example 2: Consider again the SDF graph of Figure 2. The repetitions vector of this graph is given by

$$\mathbf{q}(A, B, C, D, E) = (10, 2, 1, 1, 2). \quad (8)$$

Additionally, we have $TNSE_G((A, D)) = 10$, and $TNSE_G((B, C)) = 2$.

If a repetitions vector exists for an SDF graph, but a valid schedule does not exist, then the graph is deadlocked. Thus, an SDF graph is consistent if and only if a repetitions vector exists, and the graph is not deadlocked. For example, if we reduce the number of delays on the edge (D, E) in Figure 2 (without adding delay to any of the other edges), then the graph will become deadlocked.

In summary, SDF is currently the most widely-used dataflow model in commercial and research-oriented DSP design tools. Although SDF has limited expressive power, the model has proven to be of great practical value in the domain of signal processing and digital communication. SDF encompasses a broad and important class of applications, including modems, digital audio broadcasting systems, video encoders, multirate filter banks, and satellite

receiver systems, just to name a few [2, 11, 12, 34, 46, 51]. Commercial tools that employ SDF semantics include Simulink by The Math Works, SPW by Cadence, and ADS by Hewlett Packard. SDF-based research tools include Gabriel [32] and several key domains in Ptolemy [16], from U.C. Berkeley; and ASSIGN from Carnegie Mellon [40]. Except where otherwise noted, all of the cosynthesis techniques discussed in this chapter are applicable to SDF-based specifications.

2.3 Alternative dataflow models

To address the limited expressive power of SDF, a number of alternative dataflow models have been investigated for the specification of DSP systems. These can be divided into three major groups — the *decidable dataflow models*, which, like SDF, enable bounded memory and deadlock determination to be solved at compile time; the *dynamic dataflow models*, in which there is sufficient dynamism and expressive power that the bounded memory and deadlock problems become undecidable; and the *dataflow meta-models*, which are model-independent mechanisms for adding expressive power to broad classes of dataflow modeling approaches. Decidable dataflow models include SDF; *cyclo-static dataflow* [12] and *scalable synchronous dataflow* [44], which we discuss in Sections 2.4 and 6, respectively; and *multidimensional synchronous dataflow* [35] for expressing multidimensional DSP applications, such as those arising in image and video processing. Dynamic dataflow models include *boolean dataflow* and *integer-controlled dataflow* [14, 15], and *bounded dynamic dataflow* [41]. Metamodeling techniques relevant to dataflow include the *starcharts* approach [23], which provides flexible integration of finite state machine and dataflow models, and *parameterized dataflow* [7, 8], which provides a general mechanism for incorporating dynamic reconfiguration capabilities into arbitrary dataflow models.

2.4 Cyclo-static dataflow

Cyclo-static dataflow (CSDF) and scalable synchronous dataflow (described in Section 6) are presently the most widely-used alternatives to SDF. In CSDF, introduced by Bilsen, Engels, Lauwereins, and Peperstraete, the number of tokens produced and consumed by an actor is allowed to vary as long as the variation takes the form of a fixed, periodic pattern [12]. More precisely, each actor A in a CSDF graph has associated with it a fundamental period $\tau(A) \in Z^+$, which specifies the number of *phases* in one minimal period of the cyclic production/consumption pattern of A. For each incoming edge e of A, the scalar SDF attribute $cns(e)$ is replaced by a $\tau(A)$-tuple $(C_{e,1}, C_{e,2}, \ldots, C_{e,\tau(A)})$, where each $C_{e,i}$ is a nonnegative integer that gives the number of data values consumed from e by A in the ith phase of each period of A. Similarly, for each outgoing edge e, $prd(e)$ is replaced by a $\tau(A)$-tuple $(P_{e,1}, P_{e,2}, \ldots, P_{e,\tau(A)})$, which gives the numbers of data values produced in successive phases of A.

Example 3: A simple example of a CSDF actor is a conventional downsampler actor from multirate signal processing. Functionally, a downsampler actor (with downsampling factor N) has one incoming edge and one outgoing edge, and performs the function $y[i] = x[N(i-1)+1]$, where for $k \in Z^+$, $y[k]$ and $x[k]$ denote the kth data values produced and consumed, respectively, by the actor. Thus, for every input value that is copied to the output, $(N-1)$ input values are discarded. This functionality can be specified by a CSDF actor that has N phases. A data value is consumed from the incoming edge for all N phases, resulting in the N-component *consumption tuple* $(1, 1, \ldots, 1)$; however, a data value is produced onto the outgoing edge only on the first phase, resulting in the *production tuple* $(1, 0, \ldots, 0)$.

Like SDF, CSDF permits efficient verification of bounded memory requirements and deadlock avoidance [12]. Furthermore, static schedules can always be constructed for consistent CSDF graphs.

A CSDF actor A can easily be converted into an SDF actor A' such that if identical sequences of input data values are applied to A and A', then identical output data sequences result. Such a functionally-equivalent SDF actor A' can be derived by having each firing of A' implement one fundamental CSDF period of A (that is, $\tau(A)$ successive phases of A). Thus, for each incoming edge e' of A', the SDF parameters of e' are given by

$$del(e') = del(e);\ prd(e') = \sum_{i=1}^{\tau(A)} P_{e,i};\ \text{and similarly, } cns(e') = \sum_{i=1}^{\tau(A)} C_{e,i}, \quad (9)$$

where e is the corresponding incoming edge of the CSDF actor A.

Since any CSDF actor can be converted in this manner to a functionally equivalent SDF actor, it follows that CSDF does not offer increased expressive power at the level of individual actor functionality (input-output mappings). However, the CSDF model does offer increased flexibility in compactly and efficiently representing interactions between actors.

Example 4: As an example of increased flexibility in expressing actor interactions, consider the CSDF specification illustrated in Figure 3. This specification represents a recursive digital filter computation of the form

$$y_n = k^2 y_{n-1} + k x_n + x_{n-1}. \quad (10)$$

In Figure 3, the two-phase CSDF actor labeled A represents a scaling (multiplication) by the constant factor k. In each of its two phases, actor A consumes a data value from one of its incoming edges, multiplies the data value by k, and

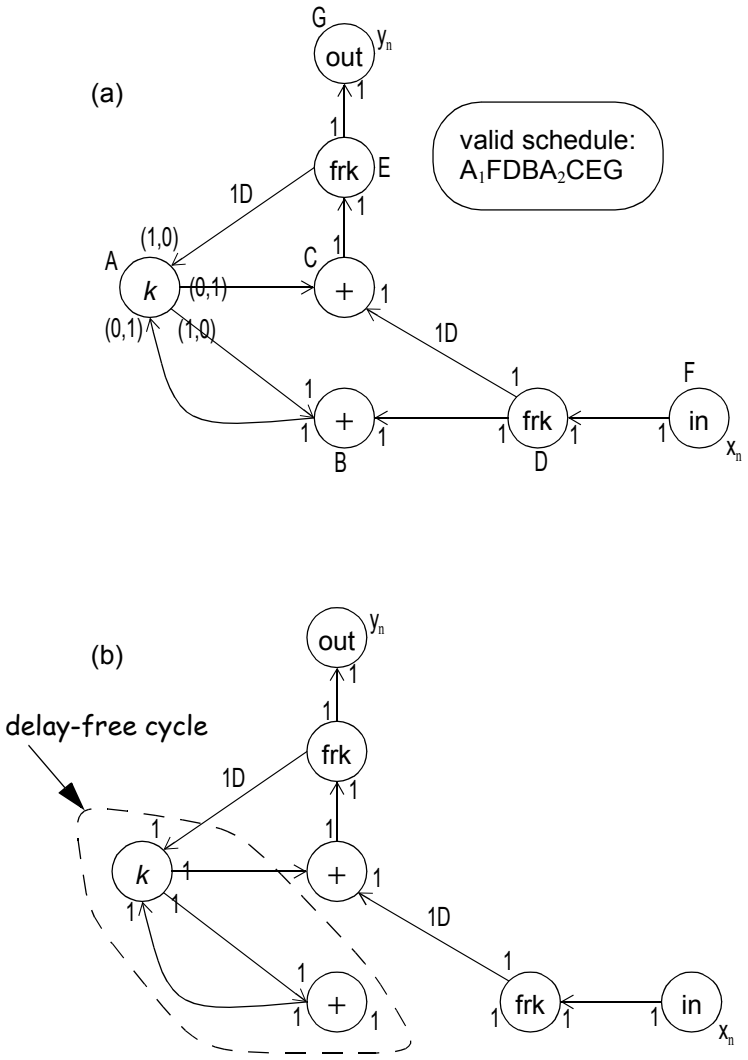

Figure 3. (a) An example that illustrates the compact modeling of resource sharing using CSDF. The actors labeled *frk* denote dataflow "forks," which simply replicate their input tokens on all of their output edges. The top right portion of the figure gives a valid schedule for this CSDF specification. Here, A_1 and A_2 denote the first and second phases of the CSDF actor A. (b) The SDF version of the specification in (a). This graph is deadlocked due to the presence of a delay-free cycle.

produces the resulting value onto one of its outgoing edges. The CSDF specification of Figure 3 thus exploits our ability to compute (10) using the equivalent formulation

$$y_n = k(ky_{n-1} + x_n) + x_{n-1}, \tag{11}$$

which requires only addition actors and k-scaling actors. Furthermore, the two k-scaling operations contained in (11) are consolidated into a single CSDF actor (actor A).

Such consolidation of distinct operations from different data streams offers two advantages. First, it leads to more compact representations since fewer vertices are required in the CSDF graph. For large or complex applications, this can result in more intuitive representations, and can reduce the time required to perform various analysis and synthesis tasks. Second, it allows a precise modeling of *resource sharing* decisions — pre-specified assignments of multiple operations in a DSP application onto individual hardware resources (such as functional units) or software resources (such as subprograms) — within the framework of dataflow. Such pre-specified assignments may arise from constraints imposed by the designer, and from decisions taken during synthesis or design space exploration.

Another advantage offered by CSDF that is especially relevant to cosynthesis tasks is that by decomposing actors into a finer level (phase-level) of specification granularity, basic behavioral optimizations such as *constant propagation* and *dead code elimination* [3, 20] are facilitated significantly [42]. As a simple example of dead code elimination with CSDF, consider the CSDF specification shown in Figure 4(a) of a multirate FIR filtering system that is expressed in terms of basic multirate building blocks. From this graph, the equivalent "acyclic precedence graph," (APG) shown in Figure 4(b), can be derived using concepts discussed in [12, 34]. In the CSDF APG, each actor corresponds to a single phase of a CSDF actor or a single firing of an SDF actor within a valid schedule. We will discuss the APG concept in more detail in Section 3.1.

From Figure 4(b), it is apparent that the results of some computations (SDF firings or CSDF phases) are never needed in the production of any of the system outputs. Such computations correspond to dead code and can be eliminated during synthesis without compromising correctness. For this example, the complete set of subgraphs that correspond to dead code is illustrated in Figure 4(b). Parks, Pino, and Lee show that such "dead subgraphs" can be detected with a straightforward algorithm [42].

Other advantages of CSDF include improved support for hierarchical specifications, and more economical data buffering [12].

In summary, CSDF is a useful generalization of SDF that maintains the properties of efficient verification, and static scheduling, while offering a more rich range of inter-actor communication patterns, and improved support for basic behavioral optimizations. CSDF concepts were introduced in the GRAPE design environment [30], which is a research tool developed at K. U. Leuven, and are currently used in a number of commercial design tools such as DSP Canvas by Angeles Design Systems, and Virtuoso Synchro by Eonic Systems.

3 Multiprocessor implementation of dataflow models

A fundamental task in synthesizing hardware and software from a data-flow specification is that of scheduling, which, as described in Section 2.2, refers to the process of determining the order in which actors will be executed. During cosynthesis, it is often desirable to obtain efficient, *parallel* implementations, which execute multiple actor firings simultaneously on different resources.

For this purpose, the class of "valid schedules" introduced in Section 2.2 is not sufficient; *multiprocessor schedules*, which consist of multiple firing

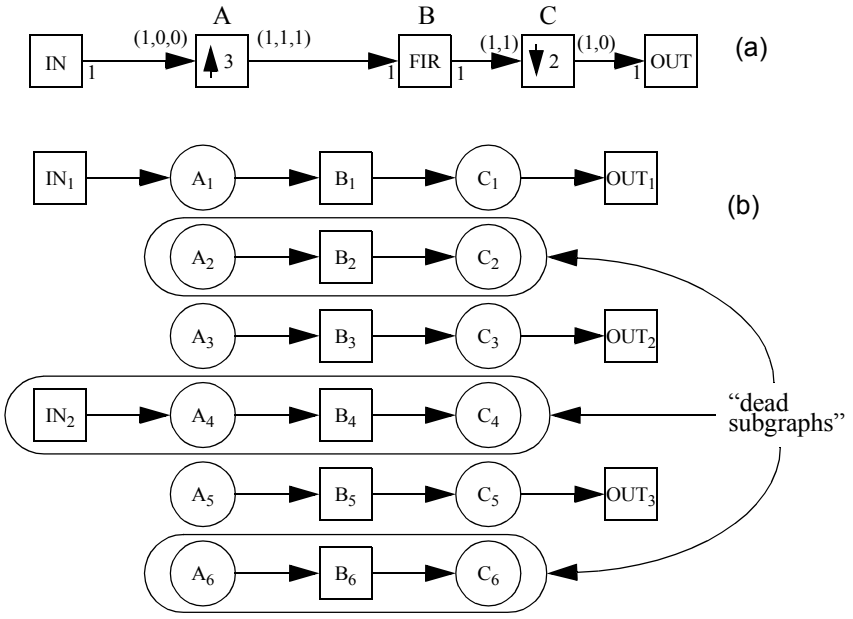

Figure 4. An example of efficient dead code elimination using CSDF.

sequences — one for each processing resource — are required. However, the consistency concepts developed in Section 2.2, are inherent to SDF specifications, and apply regardless of whether or not parallel implementation is used. In particular, when performing static, multiprocessor scheduling of SDF graphs, it is still necessary to first compute the repetitions vector, and also, to verify that the graph is deadlock-free, and the techniques for accomplishing these objectives are no different for the multiprocessor case.

However, there are a number of additional considerations that arise when attempting to construct and implement multiprocessor schedules. We elaborate on these in the remainder of this section.

3.1 Precedence expansion graphs

Associated with any connected, consistent SDF graph G, there is a unique directed graph, called its *equivalent acyclic precedence graph* (*APG*), that specifies the precedence relationships between distinct actor firings throughout an iteration of a valid schedule for G [34]. Cosynthesis algorithms typically operate on this APG representation since it fully exposes inter-firing concurrency, which is hidden in the more compact SDF representation. The APG can thus be viewed as an intermediate representation when performing cosynthesis from an SDF specification.

Each vertex of the APG corresponds to an actor firing within a single iteration period of a valid schedule. Thus, for each actor A in an SDF graph, there are $\mathbf{q}(A)$ corresponding vertices in the associated APG. For each $i = 1, 2, \ldots, \mathbf{q}(A)$, the vertex associated with the ith firing of A is often denoted as A_i. Furthermore, there is an APG edge directed from the vertex corresponding to firing A_i to the vertex corresponding to firing B_j if and only if at least one token produced by A_i is consumed by B_j.

Example 5: As a simple example, Figure 5 below shows an SDF graph and its associated APG.

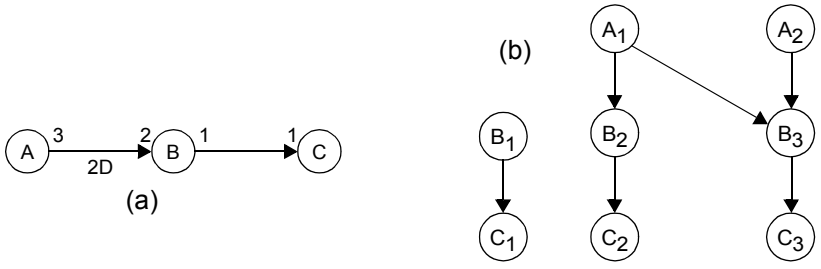

Figure 5. (a) An SDF graph, and (b) its equivalent APG.

For an efficient algorithm that systematically constructs the equivalent APG from a consistent SDF graph, we refer the reader to [47]. Similar techniques can be employed to map CSDF specifications into equivalent APG representations.

We refer to an APG representation of an SDF or CSDF application specification as a *dataflow application graph*, or simply, an *application graph.* In other words, an application graph is an application specification in which each vertex represents *exactly one* firing within a valid schedule for the graph. Additionally, when the APG is viewed in isolation (i.e., independent of any particular SDF graph), each vertex in the APG may be referred to as an *actor* without ambiguity.

3.2 Multiprocessor scheduling models

Cosynthesis requires two central tasks — *allocation* of resources (e.g., programmable processors, FPGA devices, and so-called "algorithm-based" computing modules [38]), and *scheduling* of application actors onto the allocated resources. The scheduling task can be further subdivided into three main operations: assigning actors to processors, ordering actors on each processor, and determining the time at which each actor begins execution. Based on whether these scheduling operations are performed at run-time or compile-time, we can classify multiprocessor scheduling strategies into four categories — fully static, static assignment, fully dynamic, and self-timed scheduling [31]. In fully-static scheduling, all three scheduling operations are performed at compile-time; in static allocation, only the processor assignment is performed at compile-time; and in the fully dynamic approach, all three operations are completed at run-time. As we move from fully static to fully dynamic scheduling, we trade-off simplicity and lower run-time cost for increased generality.

For DSP systems, an efficient and popular scheduling model is the *self-timed* model [31], where we obtain a fully static schedule, but we ignore the precise timing that such a strategy would enforce. Instead, processors synchronize with one another only based on interprocessor communication (*IPC*) requirements. Such a strategy retains much of the reduced overhead of fully-static scheduling; offers robustness when actor execution times are not constant or precisely known; improves efficiency by eliminating extraneous synchronization requirements; eliminates the need for specialized synchronization hardware; and naturally supports asynchronous design [31, 49]. The techniques discussed in this chapter are suitable for incorporation in the context of fully-static or self-timed scheduling.

3.3 Scheduling techniques

Numerous scheduling algorithms have been developed for multiprocessor scheduling of dataflow application graphs. Two general categories of scheduling

techniques that are frequently used in cosynthesis approaches are *clustering* and *list scheduling*.

Clustering algorithms for multiprocessor scheduling operate by incrementally constructing groupings, called *clusters*, of actors that are to be executed on the same processor. Clustering and list scheduling can be used in a complementary fashion. Typically, clustering is applied to focus the efforts of a list scheduling algorithm on effective processor assignments. When used efficiently, clustering can significantly enhance the results produced by list scheduling, and a variety of other scheduling techniques.

In *list scheduling*, a *priority list* L of actors is constructed; a global time clock c_G is maintained; and each actor T is eventually mapped into a time interval $[x_T, y_T]$ on some processor (the time intervals for two distinct actors assigned to the same processor cannot overlap). The priority list L is a linear ordering $(v_1, v_2, \ldots, v_{|V|})$ of the actors in the input application graph $G = (V, E)$ ($V = \{v_1, v_2, \ldots, v_{|V|}\}$) such that for any pair of distinct actors v_i and v_j, v_i is to be given higher scheduling priority than v_j if and only if $i < j$. Each actor is mapped to an available processor as soon as it becomes the highest-priority actor — according to L — among all actors that are *ready*. An actor is ready if it has not yet been mapped, but its predecessors have all been mapped, and all satisfy $y_T \le t$, where t is the current value of c_G. For self-timed implementation, actors on each processor are ordered according to the order of their associated time intervals.

A wide variety of actor prioritization schemes for list scheduling can be specified in terms of a *parameterized longest path function*

$$\lambda_G(A, f_v, f_e), \tag{12}$$

where $G = (V, E)$ denotes the application graph that is being scheduled; $A \in V$ is any actor in G; $f_v : V \to \overline{\mathbb{Z}}$ is a function that maps application graph actors into (extended) integers (vertex weights); and similarly, $f_e : E \to \overline{\mathbb{Z}}$ is a function that maps application graph edges into integers (edge weights). The value of $\lambda_G(A, f_v, f_e)$ is defined to be

$$max\left(\left\{\left[\sum_{i=1}^{n} f_v(snk(e_i)) + \sum_{i=1}^{n} f_e(e_i) + f_v(A)\right| \quad (e_1, e_2, \ldots, e_n) \text{ is a path in } G \text{ that originates at } A\right\}\right). \tag{13}$$

Under this formulation, the priority of an actor is taken to be the associated value of $\lambda_G(*, f_v, f_e)$; in other words, the priority list for list scheduling is constructed in decreasing order of the metric $\lambda_G(*, f_v, f_e)$.

Example 6: If actor execution times are constant, $f_v(A)$ is taken to be the execution time of A, and f_e is taken to be the *zero function on* E ($f(e') = 0$ for all $e' \in E$), then $\lambda_G(*, f_v, f_e)$ gives the famous *Hu-level* priority function [25], which is the value of the longest-cumulative-execution-time path that originates at a given actor. For homogeneous communication networks, another popular priority function is obtained by taking $f_e(e')$ to be the interprocessor communication latency associated with edge e' (the communication latency if $src(e)$ and $snk(e)$ are assigned to different processors), and again taking $f_v(A)$ to be the execution time of A. In the presence of non-deterministic actor execution times, common choices for f_v include the average- and worst-case execution times.

4 Partitioning into hardware and software

This section focuses on a fundamental component of the cosynthesis process — the partitioning of application graph actors into hardware and software. Since partitioning and scheduling are, in general, highly interdependent, these two tasks are usually performed jointly. The net result is thus an allocation (if applicable) of hardware and software processing resources and communication resources; an assignment of application graph actors to allocated resources; and a complete schedule for the derived allocation/assignment pair. Here, we examine three algorithms, ordered in increasing levels of generality, that address the partitioning problem.

4.1 GCLP

The *global criticality, local phase (GCLP) algorithm* [27], developed by Kalavade and Lee, gives an approach for combined hardware/software partitioning and scheduling for minimum latency. Input to the algorithm includes an application graph $G = (V, E)$, a target platform consisting of a programmable processor and a fabric for implementing custom hardware, and constraints on the latency, and on the code size of the software component. Each actor $A \in V$ is characterized by its execution time $t_h(A)$ and area $a_h(A)$ if implemented in hardware, and its execution time $t_s(A)$ and code size $a_s(A)$ if implemented in software. The GCLP algorithm attempts to compute a mapping of graph actors into hardware and software, and a schedule for the mapped actors. The objective is to minimize the area of the custom hardware subject to the constraints on latency and software code size.

At each iteration i of the algorithm, a ready actor is selected for mapping and scheduling based on a dynamic priority function $P_i : V \rightarrow \aleph$ that takes into

account the relative difficulty (*time-criticality*) in achieving the latency constraint based on the partial schedule S_i constructed so far. Increasing levels of time criticality translate to increased affinity for hardware implementation in the computation P_i of actor priorities. Since it incorporates the structure of the entire application graph and current scheduling state S_i, this affinity for hardware implementation is called the *global criticality.* We denote the value of global criticality computed at algorithm iteration i by $C_g(i)$.

Once a ready actor A_i is chosen for scheduling based on global criticality considerations, the hardware and software mapping alternatives for A_i are taken into account, based on so-called *local phase* information, to determine the most attractive implementation target (hardware or software) for A_i, and A_i is scheduled accordingly.

The global criticality metric $C_g(i)$ is derived by determining a tentative implementation target for each unscheduled actor in an effort to efficiently extend the partial schedule S_i into a complete schedule. The goal in this rough, schedule extension step is to determine the most economical subset H_i of unscheduled actors to implement in hardware such that the latency constraint is achieved. This subset is iteratively computed based on an actor-priority function that captures the area/time trade-offs for each actor, and a fast scheduling heuristic that computes the overall latency for a given hardware/software mapping.

Given H_i, the global criticality at iteration i is computed as an estimate of the fraction of overall computation in the set U_i of unscheduled actors that is contained in the tentatively-hardware-mapped subset H_i:

$$C_g(i) = \frac{\sum_{A \in H_i} \text{ElemOps}(A)}{\sum_{A \in U_i} \text{ElemOps}(A)}, \tag{14}$$

where $\text{ElemOps}(A)$ denotes the number of elementary operations (e.g., addition, multiplication, …) within actor A.

Once $C_g(i)$ is computed, the hardware mapping H_i is discarded, and $C_g(i)$ is loosely interpreted as an *actor-invariant probability* that any given actor will be implemented in hardware. This probabilistic interpretation is applied to compute "critical path lengths" in the application graph, in which the implementation targets, and hence the execution times, of unscheduled actors are not yet known. More specifically, the actor that is selected for mapping and scheduling at algorithm iteration i is chosen to be one (ties are broken arbitrarily) that maximizes

$$\lambda_G(A, \tau_i, \varepsilon_0) \tag{15}$$

over all $A \in \text{Ready}(S_i)$, where λ_G is the parameterized longest path function defined by (13); $\tau_i : V \rightarrow \aleph$ is defined by

$$\tau_i(X) = C_g(i)t_h(X) + (1 - C_g(i))t_s(X); \tag{16}$$

$\varepsilon_0 : E \rightarrow \{0\}$ is the zero function on E; and $\text{Ready}(S_i)$ is the set of application graph actors that are ready at algorithm iteration i. The "execution time estimate" given in (16) can be interpreted loosely as the expected execution time of actor A if one wishes to extend the partial schedule S_i into an economical implementation that achieves the given latency constraint.

4.1.1 Hardware/software selection threshold

In addition to determining (via (16)) the actor A_i that is to be scheduled at algorithm iteration i, the global criticality $C_g(i)$ is used to determine whether A_i should be implemented in hardware or software. In particular, an actor-dependent cut-off point $threshold(A_i)$ is computed such that if $C_g(i) \geq threshold(A_i)$, then A_i is mapped into hardware or software based on the alternative that results in the earliest completion time for A_i (based on the partial schedule S_i), while if $C_g(i) < threshold(A_i)$, then the mapping for A_i is chosen to be the one that results in the leanest resource consumption.

The objective function selection threshold associated with an actor A_i is computed as

$$threshold(A_i) = 0.5 + LocalPhaseDelta(A_i), \tag{17}$$

where $LocalPhaseDelta(A_i)$ measures aspects of the specific hardware/software trade-offs associated with actor A_i. More specifically, this metric incorporates the classification of A_i as either an *extremity* actor, a *repeller* actor, or a "normal" actor. An extremity actor is either a software extremity or a hardware extremity. Intuitively, a software extremity is an actor whose software execution time (SET) is one of the highest SETs among all actors, but whose hardware implementation area (HIA) is *not* among the highest HIAs. Similarly, a hardware extremity is an actor whose HIA is one of the highest HIAs, but whose SET is *not* among the highest SETs. The precise methods to compute thresholds that determine the classes of "highest" SET and HIA values are parameters of the GCLP framework that are to be configured by the tool developer or the user.

An actor is a *repeller* with respect to software (hardware) implementation if it is not an extremity actor, and its functionality contains components that are distinguishably ill-suited to efficient software (hardware) implementation. For example the *bit-level instruction mix*, defined as the overall proportion of bit-

level operations, has been identified as an actor property that is useful in identifying software repellers (a *software repeller property*). Similarly, the proportion of memory-intensive instructions is a hardware repeller property. For each such repeller property of a given repeller actor, a numeric estimate is computed to characterize the degree to which the property favors software or hardware implementation for the actor.

The *LocalPhaseDelta* value in (17) is set to zero for normal actors — i.e., actors that are neither extremity nor repeller actors. For extremity actors, the value is determined as a function of the SETs and HIAs, and for repeller actors it is computed as

$$LocalPhaseDelta(A_i) = \frac{1}{2}(\varphi_h - \varphi_s), \tag{18}$$

where φ_h and φ_s represent normalized, weighted sums of contributions from individual hardware and software repeller properties, respectively. Thus, for example, if the hardware repeller properties of actor A_i dominate ($\varphi_h > \varphi_s$), it becomes more likely (from (17) and (18)), that $(C_g(i) < threshold(A_i))$, and thus, that A_i will be mapped to software (assuming that the communication and code size costs associated with software mapping are not excessive).

The overall appeal of the GCLP algorithm stems from its ability to integrate global, application- and partial-schedule-level information with the actor-specific, heterogeneous-mapping metrics associated with the local phase concept. Also, the scheduling, estimation, and mapping heuristics within the GCLP algorithm consider area and latency overheads associated with communication between hardware and software. Thus, the algorithm jointly considers actor execution times, hardware and software capacity costs, and both temporal and spacial costs associated with interprocessor communication.

4.1.2 Cosynthesis for multi-function applications

Kalavade and Subrahmanyam have extended the GCLP algorithm to handle cosynthesis involving multiple applications that are operated in a time-multiplexed manner [28]. Such *multi-function* systems arise commonly in embedded applications. For example, a video encoding system may have to be designed to support a variety of formats, such as MPEG2, H.261, and JPEG, based on the different modes of operation that are available to the user.

The *multi-application codesign problem* is a formulation of multi-function cosynthesis in which the objective is exploit similarities between distinct system functions to streamline the result of synthesis. An instance of this problem can be viewed as a finite set of inputs to the original GCLP algorithm described earlier in this section. More precisely, an instance of multi-application codesign consists

of a set of application graphs $appset = \{G_1, G_2, \ldots, G_N\}$, where each $G_i = (V_i, E_i)$ has an associated latency constraint L_i. Furthermore, if we define $V_{\text{appset}} = (V_1 \cup V_2 \cup \ldots \cup V_N)$, then each actor $A \in V_{\text{appset}}$ is characterized by its *node type* $type(A)$, execution time $t_h(A)$ and area $a_h(A)$ if implemented in hardware, and execution time $t_s(A)$ and code size $a_s(A)$ if implemented in software. The objective is to construct an assignment of actors in V_{appset} into hardware and software, and schedules for all of the application graphs in *appset* such that the schedule for each G_i satisfies its associated latency constraint L_i, and overall hardware area is minimized. An underlying assumption in this codesign problem is that at any given time during operation, at most one of the application graphs in *appset* may be active.

The node type attribute specifies the function class of the associated actor, and is used to identify opportunities for resource sharing across multiple actors within the same application, as well as across actors in different applications. For example, if two applications graphs each contain a DCT module (an actor whose node type is that of a DCT), and one of these is mapped to hardware, then it may be profitable to map the other DCT actor into hardware as well, especially since both DCT actors will never be active at the same time.

4.1.3 Modified threshold adjustment

Kalavade's "multi-function extension" to GCLP, which we call *GCLP-MF*, retains the global criticality concept, and the threshold-based approach to mapping actors into hardware and software. However, the metrics associated with local phase computation (threshold adjustment), are replaced with a number of alternative metrics, called *commonality measures*, that take into account characteristics that are relevant to the multi-function case. These metrics are consistently normalized to keep their values within predictable and meaningful ranges.

Recall that higher values of the GCLP threshold favor software implementation, while lower values favor hardware implementation, and the threshold in GCLP is computed from (17) as the sum of 0.5 and an adjustment term, called the local phase. In GCLP, this local phase adjustment term is replaced by an alternative function that incorporates re-use of node types across different actors and applications, and actor-specific, performance-area trade-offs. Type re-use is quantified by a *type repetitions* metric, denoted R, which gives the total number of actor instances of a given type over all application graphs in *appset*. In other words, for a given node type θ,

$$R(\theta) = \sum_{(V, E) \in appset} \left| \{A \in V \mid (type(A) = \theta)\} \right|, \tag{19}$$

and the normalized form of this metric, which we denote R_N, is defined by nor-

malizing to values restricted within $[0, 1]$:

$$R_N(\theta) = \frac{R(\theta)}{max(\{R(type(A))|(A \in V_{appset})\})}. \quad (20)$$

Performance-area trade-off information is quantified by a metric T that measures the speedup in moving an actor implementation from software to hardware relative to the required hardware area:

$$T(A) = \frac{t_s(A) - t_h(A)}{a_h(A)} \text{ for each } A \in V_{appset}. \quad (21)$$

The normalized form of this metric, T_N, is defined in a fashion analogous to (20) to again obtain a value within $[0, 1]$.

4.1.4 GCLP-MF algorithm versions

Two versions of GCLP-MF have been proposed. In the first version, which we call *GCLP-MF-A*, the normalized commonality metrics R_N and T_N are combined into a composite metric κ, based on user-defined weighting factors α_1 and α_2:

$$\kappa(A) = \alpha_1 R_N(type(A)) + \alpha_2 T_N(A) \text{ for each } A \in V_{appset}. \quad (22)$$

This composite metric, in turn, is mapped into a $[0, 0.5]$-normalized form by applying a formula analogous to (20), and then multiplying by 0.5. The resulting normalized, composite metric, which we denote by κ_N becomes the threshold adjustment value for GCLP-MF-A. More specifically, in GCLP-MF-A, the hardware/software mapping threshold is computed as

$$threshold(A) = 0.5 - \kappa_N(A). \quad (23)$$

This threshold value, which replaces the original GCLP threshold expression (17), is compared against an actor's application-specific global criticality measure during cosynthesis. Intuitively, this threshold systematically favors hardware implementation for actor types that have relatively high type-repetition counts, and for actors that deliver large hardware vs. software performance gains with relatively small amounts of hardware area overhead.

The GCLP-MF-A algorithm operates by applying to each member of *appset* the original GCLP algorithm with the threshold computation (17) replaced by (23).

The second version, *GCLP-MF-B*, attempts to achieve some amount of "interaction" across cosynthesis decisions of different application graphs in *appset* rather than processing each application in isolation. In particular, the

composite adjustment term (22) is discarded, and instead, a mechanism is introduced to allow cosynthesis decisions for the most difficult (from a synthesis perspective) applications to influence those that are made for less difficult applications. The difficulty of an application graph $G_i \in appset$ is estimated by its *criticality*, which is defined to be the sum of the software execution times divided by the latency constraint.

$$criticality(G_i) = \frac{\sum_{v \in V_i} t_s(v)}{L_i}. \quad (24)$$

Intuitively, an application with high criticality requires a large amount of hardware area to satisfy its latency constraint, and thus makes it more difficult to meet the minimization objective of cosynthesis.

GCLP-MF-B operates by processing application graphs in decreasing order of their criticality, keeping track of inter-application resource-sharing possibilities throughout the cosynthesis process, and systematically incorporating these possibilities into the hardware/software selection threshold. Resource sharing information is effectively stored as an actor-indexed array S of 3-valued "sharing state" elements. For a given actor A, $S[A] = \text{NULL}$ indicates that no actor of type $\text{type}(A)$ has been considered in a previous mapping step; $S[A] = \text{HW}$ indicates that a $\text{type}(A)$ actor has previously been considered, and has been mapped into hardware; and $S[A] = \text{SW}$ indicates a previous software mapping decision for $\text{type}(A)$.

Like GCLP-MF-A, the GCLP-MF-B algorithm, applies the original GCLP algorithm to each application graph separately with a modification of the hardware/software threshold function (17). Specifically, the threshold in GCLP-MF-B is computed as

$$\text{threshold}(A) = \begin{cases} (0.5 - T_N(A)) \text{ if } (S[A] = \text{NULL}) \\ (0.5 - R_N(A)) \text{ if } (S[A] = \text{HW}) \\ (0.5 + R_N(A)) \text{ if } (S[A] = \text{SW}) \end{cases}. \quad (25)$$

Thus, previous mapping decisions (from equal- or higher-criticality applications), together with commonality metrics, are used to determine whether or not a given actor is mapped into hardware or software.

Experimental results have shown that for multi-function systems, both versions of GCLP-MF significantly outperform isolated applications of the original GCLP algorithm to the application graphs in $appset$, and that version B, which incorporates the commonality metrics used in A in addition to the shared

mapping state S, outperforms version A.

4.2 COSYN

Optimal or nearly-optimal hardware/software cosynthesis solutions are difficult to achieve since there are numerous relevant implementation considerations and constraints. The COSYN algorithm [18], developed by Dave, Lakshminarayana, and Jha, takes numerous elements of this complexity into account. The design considerations and objectives addressed by the algorithm include allowing arbitrary, possibly heterogeneous collections of processors and communication links; intraprocessor concurrency (e.g., in FPGAs and ASICs); preemptive vs. non-preemptive scheduling; actor duplication on multiple processors to alleviate communication bottlenecks; memory constraints; average, quiescent and peak power dissipation in processing elements and communication links; latency (in the form of actor deadlines); throughput (in the form of subgraph initiation rates); and overall dollar cost, which is the ultimate minimization objective.

4.2.1 Algorithm flow

Input to the COSYN algorithm includes an application graph $G = (V, E)$ that may consist of several independent subgraphs that operate at different rates (periods) and with different deadlines; a library of processing elements $R = \{r_1, r_2, \ldots, r_m\}$; a set of communication resources ("links") $C = \{c_1, c_2, \ldots, c_n\}$; an actor execution time function $t_e : V \times R \rightarrow \overline{\aleph}$, which specifies the execution time of each actor on each candidate processing resource; a communication time function $t_c : E \times C \rightarrow \overline{\aleph}$, which gives the latency of communication of each edge on each candidate communication resource; and a deadline function $deadline : V \rightarrow \overline{\aleph}$, which specifies an optional maximum allowable completion time for each actor. Under this notation, an infinite value of t_e (t_c) indicates an incompatibility between the associated actor/resource (edge/resource) pair, and similarly, $deadline(v) = \infty$ if there is no deadline specified for actor v.

The overall flow of the COSYN algorithm is outlined in Figure 6. In the initial *FormClusters* phase, the application graph is analyzed to identify subgraphs that are to be grouped together during the allocation and assignment exploration phases. After clusters have been formed, they are examined — one by one — and allocated by exploring their respective ranges of possible allocations, and selecting the ones that best satisfy certain criteria that relate to the given performance and cost objectives. As individual allocation decisions are made, execution times of actors in the associated clusters become fixed, and this information is used to re-evaluate cluster priorities for future cluster selection decisions, and also to re-evaluate actor edge priorities during scheduling (to eval-

uate candidate allocations). Thus, cluster selection and scheduling decisions are computed dynamically based on all previously committed allocations.

4.2.2 Cluster formation

Cluster decisions during the *FormCluster* phase are guided by a metric that prioritizes actors based on deadline- and communication-conscious critical path analysis. Like cluster selection and allocation decisions, actor priorities for clustering are dynamically evaluated based on all previous clustering operations. The priority of an actor for clustering is computed as

$$\lambda_G(A, f_t, f_c), \tag{26}$$

where the execution time contribution function $f_t : V \rightarrow \overline{\mathbb{Z}}$ is given as the worst case execution time offset by the actor deadline —

$$f_t(v) = max(\{t_e(v, r_i) \mid (r_i \in R) \textbf{ and } (t_e(v, r_i) < \infty)\}) - deadline(v); \tag{27}$$

and the communication time contribution function $f_c : E \rightarrow \aleph$ is given as the worst case communication cost, based on all previous clustering decisions —

$$f_c(e, c) = \begin{cases} 0 \text{ if } (e \in subsumed) \\ max(\{t_c(e, c_i) \mid (c_i \in C) \textbf{ and } (t_c(e, c_i) < \infty)\}), \text{ otherwise} \end{cases}. \tag{28}$$

Here, *subsumed* denotes the set of edges in E that have been "enclosed" by the clusters created by all previous clustering operations; that is, the set of edges e such that $src(e)$ and $snk(e)$ have already been clustered, and both belong to the same cluster.

```
function COSYN
    FormClusters(G) → Cluster set X
    unallocated = X
    for i = 1, 2, …, |X|
        ComputeClusterPriorities(unallocated)
        Select a maximum priority cluster C_i ∈ unallocated
        Evaluate possible allocations for C_i and select best one
        unallocated = unallocated – {C_i}
    end for
end function
```

Figure 6. A pseudocode sketch of the COSYN algorithm.

At each clustering step, an unclustered actor A that maximizes $\lambda_G(*, f_t, f_c)$ is selected, and based on certain compatibility criteria, A is first either merged into the cluster of a predecessor actor, or inserted into a new cluster, and then the resulting cluster is may be further expanded to contain a successor of A.

4.2.3 Cluster allocation

After clustering is complete, we have a disjoint set of clusters $X = \{X_1, X_2, \ldots, X_p\}$, where each X_i represents a subset of actors that are to be assigned to the same physical processing element. Clusters are then selected one at a time, and for each selected cluster, the possible allocations are evaluated by scheduling. At each cluster selection step, a cluster with maximal priority (among all clusters that have not been selected in previous steps) is selected, where the priority of a cluster is simply taken to be the priority of its highest-priority actor, and actor priorities are determined using an extension of (26) that takes into account the effects of any previously-committed allocation decisions. More precisely, we suppose that for each edge $e \notin subsumed$, $asgn(e) = \text{NULL}$ if e has not yet been assigned to a communication resource, and otherwise, $asgn(e) \in C$ gives the resource type of the communication link to which e has been assigned. Similarly, we allow a minor abuse of notation, and suppose that for each actor A, $asgn(A) = \text{NULL}$ if e has not yet been assigned to a processing element (i.e., the enclosing cluster has not yet been allocated), and otherwise, $asgn(A) \in R$ gives the resource type of the processing element to which e has been assigned. Actor priority throughout the cluster allocation phase of COSYN is then computed as

$$\lambda_G(A, g_t, g_c), \tag{29}$$

where $g_t : V \to \overline{\mathbb{Z}}$ is defined by

$$g_t(v) = \begin{cases} f_t(v) \text{ if } (asgn(v) = \text{NULL}) \\ t_e(v, asgn(v)) - deadline(v), \text{ otherwise} \end{cases}, \tag{30}$$

and similarly, $g_c : E \to \aleph$ is defined by

$$g_c(e) = \begin{cases} f_c(e) \text{ if } (asgn(e) = \text{NULL}) \\ t_c(e, asgn(e)), \text{ otherwise} \end{cases}. \tag{31}$$

In other words, if an actor or edge x has been assigned to a resource r, x is modeled with the latency of x on the resource type associated with r, and otherwise, the worst case latency is used to model x.

As clusters are allocated, the values $\{asgn(x) | x \in (E \cup V)\}$ change, in general, and thus, for improved accuracy, actor priorities are re-evaluated — using (29), (30), (31) — during subsequent cluster allocation steps.

4.2.4 Allocation selection

After a cluster is selected for allocation, candidate allocations are evaluated by scheduling and *finish time estimation*. During scheduling, actors and edges are processed in an order determined by their priorities, and considerations such as overlapped vs. non-overlapped communication, and actor preemption are taken into account at this time. Once scheduling is complete, the best and worst case finish times of the actors and edges in the application graph are estimated — based on their individual best and worst cases latencies — to formulate an overall evaluation of the candidate allocation.

The best and worst case latencies associated with actors and edges are determined in a manner analogous to the "allocation-conscious" priority contribution values $g_t(*)$ and $g_c(*)$ computed in (30) and (31). For each actor $v \in V$, the best case latency is defined by

$$t_{\text{best}}(v) = \begin{cases} min(\{t_e(v, r_i) | (r_i \in R)\}) \text{ if } (asgn(v) = \text{NULL}) \\ t_e(v, asgn(v)), \text{ otherwise} \end{cases}, \quad (32)$$

and similarly, the best case latency for each edge $e \in E$ is defined by

$$t_{\text{best}}(e) = \begin{cases} min(\{t_c(e, c_i) | (c_i \in C)\}) \text{ if } (asgn(e) = \text{NULL}) \\ t_c(e, asgn(e)), \text{ otherwise} \end{cases}. \quad (33)$$

The worst case latencies, denoted $t_{\text{worst}}(v)$ and $t_{\text{worst}}(e)$, are defined (using the same minor abuse of notation) in a similar fashion.

From these best and worst case latencies, allocation-conscious best and worst case *finish* time estimates F_{best} and F_{worst} of each actor and each edge are computed by

$$F_{\text{best}}(v) = max(\{F_{\text{best}}(e_{\text{in}}) + t_{\text{best}}(v) | e_{\text{in}} \in in(v)\}), \text{ and} \quad (34)$$

$$F_{\text{worst}}(v) = max(\{F_{\text{worst}}(e_{\text{in}}) + t_{\text{worst}}(v) | e_{\text{in}} \in in(v)\}) \text{ for } v \in V; \quad (35)$$

$$F_{\text{best}}(e) = F_{\text{best}}(src(e)) + t_{\text{best}}(e), \text{ and} \quad (36)$$

$$F_{\text{worst}}(e) = F_{\text{worst}}(src(e)) + t_{\text{worst}}(e) \text{ for } e \in E. \quad (37)$$

The worst case and best case finish times, as computed by (34)-(37), are

used in evaluating the quality of a candidate allocation. Let $V_{\text{deadline}} \subseteq V$ denote the subset of actors for which deadlines are specified; let α denote the set of candidate allocations for a selected cluster; and let $\alpha' \subseteq \alpha$ be the set of candidate allocations for which all actors in V_{deadline} have their corresponding deadlines satisfied in the best case (i.e., according to $\{F_{\text{best}}(v)\}$). If $\alpha' \neq \varnothing$, then an allocation is chosen from the subset α' that *maximizes* the sum

$$\sum_{v \in V_{\text{deadline}}} F_{\text{worst}}(v) \tag{38}$$

of worst-case finish times over all actors for which pre-specified deadlines exist. On the other hand, if $\alpha' = \varnothing$, then an allocation is chosen from α that *maximizes* the sum

$$\sum_{v \in V_{\text{deadline}}} F_{\text{best}}(v) \tag{39}$$

of best-case finish times over all actors for which deadlines exist. In both cases, the maxima over the respective sets of sums are taken because they ultimately lead to final allocations that have lower overall dollar cost [18].

4.2.5 Accounting for power consumption

A "low power version" of the COSYN algorithm, called COSYN-LP, has been developed to minimize power consumption along with overall dollar cost. In addition to the algorithm inputs defined in Section 4.2.1, COSYN LP also employs *average power dissipation functions* $p_e : V \times R \rightarrow \overline{\aleph}$ *and* $p_c : E \times C \rightarrow \overline{\aleph}$. The value of $p_e(v, r_i)$ gives an estimate of the average power dissipated while actor v executes on processing resource r_i, and similarly, the value of $p_c(e, c_i)$ estimates the average power dissipated when edge e executes on communication resource c_i. Again, infinite values in this notation correspond to incompatibility relationships between operations (actors or edges) and resource types. Similar functions are also defined for peak (maximum instantaneous) power consumption, and quiescent power consumption (power consumption during periods of inactivity) of resources for processing and communication.

COSYN-LP incorporates modifications to the clustering and allocation evaluation phases that take actor and edge power consumption information into account. For example, the cluster formation process is modified to use the following power-oriented actor priority function:

$$\lambda_G(A, \rho_t, \rho_c), \tag{40}$$

Here, $\rho_t : V \rightarrow \aleph$ is defined by

$$\rho_t(v) = t_e(v, r_{\text{worst}}(v)) \times p_e(v, r_{\text{worst}}(v)), \tag{41}$$

where $r_{\text{worst}}(v)$ is a processing resource type that maximizes the execution time $t_e(v, *)$ of v; and similarly, $\rho_c : E \rightarrow \aleph$ is defined by

$$\rho_c(e) = t_c(e, c_{\text{worst}}(e)) \times p_c(e, c_{\text{worst}}(e)), \tag{42}$$

where $c_{\text{worst}}(e)$ is a communication resource type that maximizes the communication latency $t_c(e, *)$ of e. There is slight ambiguity here since there may be more than one processing (communication) resource that maximize the latency for a given actor (edge); tie breaking in such cases can be performed arbitrarily (it is not specified as part of the algorithm).

Thus, in COSYN-LP, priorities for cluster formation are computed on the basis of average power dissipation based on worst-case execution times.

In a similar manner, the average power dissipation metrics, along with the peak and quiescent power metrics are incorporated into the cluster allocation phase of COSYN-LP. For details, we refer the reader to [18].

4.3 CodeSign

As part of the CodeSign project at ETH Zurich, Blickle, Teich, and Thiele have developed a search technique for hardware/software cosynthesis [13] that is based on the framework of *evolutionary algorithms.* In evolutionary algorithms, complex search spaces are explored by encoding candidate solutions as "chromosomes," and evolving "populations" of these chromosomes by applying the principles of *reproduction* (retention of chromosomes in a population), *crossover* (derivation of new chromosomes from two or more "parent" chromosomes), *mutation* (modification of individual chromosomes), and *fitness* (metrics for evaluating the quality of chromosomes) [5]. These principles incorporate probabilistic techniques to derive new chromosomes from an existing population, and to replace portions of a population with selected, newly-derived chromosomes.

4.3.1 Specifications

A key innovation in the CodeSign approach is a novel formulation of joint allocation, assignment, and scheduling as mappings between sequences of graphs, and "activations" of vertices and edges in these graphs. This formulation is intuitively appealing, and provides a natural encoding structure for embedding within the framework of evolutionary algorithms.

The central data structure that underlies the CodeSign cosynthesis formulation is the *specification.* A CodeSign specification can be viewed as an ordered pair $S = (H_S, M_S)$, where $H_S = \{G_1, G_2, \ldots, G_N\}$; each G_i is a directed graph (called a "dependence graph") (V_i, E_i); and each M_i is a set of *mapping edges*

that connect vertices in successive dependence graphs — that is, for each $e \in M_i$, $src(e) \in V_i$ and $snk(e) \in V_{i+1}$. If the specification in question is understood, we write

$$V_H = \bigcup_{i=1}^{N} V_i,\ E_H = \bigcup_{i=1}^{N} E_i, \text{ and } E_M = \bigcup_{i=1}^{N-1} M_i. \tag{43}$$

Thus, V_H and E_H denote the sets of all dependence graph vertices and edges, respectively, and E_M denotes the set of all mapping edges. The *specification graph* of S is the graph $G_S = (V_S, E_S)$ obtained by integrating all of the dependence graphs and mapping edges: $V_S = V_H$, and $E_S = (E_H \cup E_M)$.

The "top-level" dependence graph (the *problem graph*) G_1 gives a behavioral specification of the application to be implemented. In this sense, it is similar to the application graph concept defined in Section 3.1. However, it is slightly different in its incorporation of special *communication vertices* that explicitly represent inter-actor communication, and are ultimately mapped onto communication resources in the target architecture [13].

The remaining dependence graphs $G_2, G_3, \ldots, G_N$ specify different levels of abstraction or refinement during implementation. For example, a dependence graph could specify an architectural description consisting of available resources for computation and communication (*architecture graph*), and another dependence graph could specify the decomposition of a target system into integrated circuits and off-chip buses (*chip graph*). Due to the general nature of the CodeSign specification formulation, there is full flexibility to define alternative or additional levels of abstraction in this manner.

Dependence graph edges specify connectivity between modules within the same level of abstraction, and mapping edges specify *compatibility* relationships between successive abstraction levels in a specification. That is, $e \in E_M$ indicates that $src(e)$ "can be implemented by" $snk(e)$.

Example 7: Figure 7(a) provides an illustration of a CodeSign specification for hardware/software cosynthesis onto an architecture that consists of a programmable processor resource P_S, a resource for implementing custom hardware P_H, and a bidirectional bus B that connects these two processing resources. The v_i s denote problem graph actors, and the c_i s denote communication vertices. Here, only hardware implementation is allowed for v_2 and v_5; only software implementation is allowed for v_3; and v_1, v_4 may each be mapped to either hardware or software. Thus, for example, there is no edge connecting v_2 or v_5 with the vertex P_S associated with the programmable processor. In general, communication vertices can be mapped either to the bus B (if the source and sink vertices

are mapped to different processing resources) or *internally* to either the hardware (P_H) or software (P_S) resource (if the source and sink are mapped to the same processing resource). However, mapping restrictions of the problem graph actors may limit the possible mapping targets of a communication vertex. For example, since v_2 and v_3 are restricted, respectively, to hardware and software implementation, communication vertex c_2 must be mapped to the bus B. Similarly, c_3 can be mapped to P_S or B, but not to P_H. The set of mapping edges for this example is given by

$$E_M = \{(v_1, P_H), (v_1, P_S), (v_2, P_H), (v_3, P_S), (v_4, P_H), (v_4, P_S), (v_5, P_H), (c_1, B), (c_1, P_S), (c_2, B), (c_3, B), (c_3, P_S), (c_4, B)\} \quad (44)$$

4.3.2 Activation functions

Allocations and assignments of specification graphs are formulated in terms of *activation functions*. An activation function for a specification graph G_S is any function $a : (V_S \cup E_S) \rightarrow \{0, 1\}$ that maps vertices and edges of G_S into binary numbers. If $x \in (V_S \cup E_S)$ is a vertex or a dependence graph edge, then $a(x) = 1$ is equivalent to the *use* or *instantiation* of x in the associated allocation. On the other hand, if x is a mapping edge, then $a(x) = 1$ if and only if $src(x)$ is implemented by $snk(x)$ according to the associated assignment.

Thus, an activation function uniquely determines an allocation and assignment for the associated specification. The allocation associated with an activation function a can be expressed in precise terms by

$$\alpha(a) = \{x \in (V_H \cup E_H) \,|\, a(x) = 1\}, \quad (45)$$

and similarly, the assignment associated with a is defined by

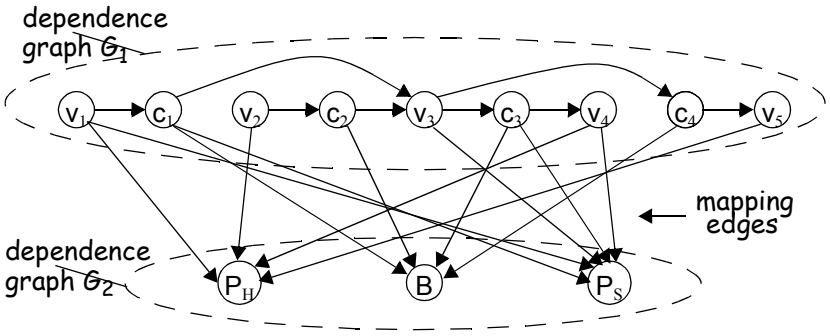

Figure 7. An illustration of a specification in CodeSign.

$$\beta(a) = \{e \in E_M | a(e) = 1\}. \quad (46)$$

We say that $x \in (V_S \cup E_S)$ is *activated* if $a(x) = 1$. The allocation and assignment associated with an activation function a are *feasible* if for each activated mapping edge $e \in \beta(a)$, the source and sink vertices are activated — that is, $src(e), snk(e) \in \alpha(a)$; for each activated vertex $v \in \alpha(a)$, there exists exactly one activated, output mapping edge $mapping(v)$ — that is, $|(out(v) \cap \beta(a))| = 1$; and for each activated dependence graph edge $e \in \alpha(a)$, either

$$mapping(src(e)) = mapping(snk(e)), \text{ or}$$
$$(mapping(src(e)), mapping(snk(e))) \in \alpha(a). \quad (47)$$

This last condition, (17), simply states that $src(e)$ and $snk(e)$ must either be assigned to same vertex in the succeeding dependence graph, or there must be an activated edge that provides the appropriate communication between the distinct vertices that $src(e)$ and $snk(e)$ are mapped to.

4.3.3 Evolutionary algorithm approach

The overall approach in the CodeSign synthesis algorithm is to encode allocation and assignment information in the chromosome data structure of the evolutionary algorithm, and use a deterministic heuristic for scheduling, since effective deterministic techniques exist for computing schedules given pre-specified allocations and assignments [20].

Decoding of a chromosome (e.g., to evaluate its fitness) begins by interpreting the allocation (activation) status (0 or 1) of each specification graph vertex that is given in the chromosome. Some allocations obtained in this way may be "incomplete" in the sense that there may be some functional vertices for which no compatible resources are instantiated. Such incompleteness in allocations is "repaired" by activating additional vertices based on a *repair allocation priority list*, which is also a component of the chromosome due to the relatively large impact of resource activation decisions on critical implementation metrics, such as performance and area. This priority list specifies the order in which vertices will be considered for activation during repair of allocation incompleteness.

After a chromosome has been converted into its associated allocation, and incompleteness of the allocation has been repaired, the assignment information from the chromosome is decoded. The coding convention for assignment information has been carefully devised to be orthogonal to the allocation encoding, so that the process of interpreting assignment information is independent of the given allocation. This independence between the interpretation of allocation and assignment information is important in facilitating efficient evolution of the chro-

mosome population [13].

Like allocation repair information, assignment information is encoded in the form of priority lists — each dependence graph vertex has an associated priority list $L_\beta(v)$ of its outgoing mapping edges $(out(v) \cap E_M)$. These priority lists are interpreted by examining each allocated vertex v, and activating the first member of $L_\beta(v)$ that does not conflict with the requirements of a feasible allocation/assignment that were discussed in Section 4.3.2

It is possible that a feasible allocation/assignment does not result from the decoding of a particular chromosome. Indeed, Blickle has shown that the problem of determining a feasible allocation/assignment is computationally intractable [13], so straightforward techniques — such as applying the decoding process to random chromosomes — cannot be relied upon to consistently achieve feasiblity.

If such infeasibility is determined during the decoding process, then a significant *penalty* is incorporated into the fitness of the associated chromosome. Otherwise, the decoded allocation and assignment are scheduled using a deterministic scheduling heuristic, and the resulting schedule, along with the assignment and allocation, are assessed in the context of the designer's optimization constraints and objectives to determine the chromosome fitness.

In summary, the CodeSign cosynthesis algorithm incorporates a novel *specification graph* data structure, and an evolutionary algorithm formulation that encodes allocation and assignment information in terms of specification graph concepts. Due to space limitations, we have suppressed several interesting details of the complete synthesis algorithm, including mechanisms for promoting resource sharing, and details of the scheduling heuristic. The reader is encouraged to consult [13] for a comprehensive discussion.

5 Synchronization optimization

In Section 3.2, we discussed the utility of self-timed multiprocessor implementation strategies in the design of efficient and robust parallel processing engines for DSP. For self-timed DSP multiprocessors, an important consideration in addition to hardware/software partitioning, and the associated scheduling task, is *synchronization* to ensure the integrity of *interprocessor communication operations* associated with dataflow edges whose source and sink actors are mapped to different processing elements. Since cost is often a critical constraint, embedded multiprocessors must often use simple communication topologies, and limited, if any, hardware support for synchronization. A variety of efficient techniques have been developed to optimize synchronization for such cost-constrained, self-timed multiprocessors [9, 10, 49]. Such techniques can significantly reduce the execution time and power consumption overhead associated with syn-

chronization, and can be used as post-processing steps to any of the partitioning algorithms discussed in Section 4, as well as to a wide variety of multiprocessor scheduling algorithms for dataflow graphs, such as those described in [19, 24, 47].

In this section, we present an overview of these approaches to synchronization optimization. Specifically, we discuss two closely-related graph-theoretic models, the *IPC graph* G_{ipc} [48] and the *synchronization graph* G_{s} [9], that are used to model the self-timed execution of a given parallel schedule for an application graph, and we discuss the application of these models to the systematic streamlining of synchronization functionality.

Given a self-timed multiprocessor schedule for an application graph G, we derive G_{ipc} and G_{s} by first instantiating a vertex for each actor, connecting an edge from each actor to the actor that succeeds it on the same processor, and adding an edge that has unit delay from the last actor on each processor to the first actor on the same processor. Also, for each edge (x, y) in G that connects actors that execute on different processors, an *IPC edge* is instantiated in G_{ipc} from x to y. Figure 8(c) shows the IPC graph that corresponds to the application graph of Figure 8(a), and the processor assignment and actor ordering of Figure 8(b).

Each edge in G_{ipc} and G_{s} is either an *intraprocessor edge* or an *interprocessor edge*. Intraprocessor edges model the ordering (specified by the given parallel schedule) of actors assigned to the same processor; interprocessor edges in G_{ipc}, called *IPC edges*, connect actors assigned to distinct processors that must communicate for the purpose of data transfer; and interprocessor edges in G_{s}, called *synchronization edges*, connect actors assigned to distinct processors that must communicate for synchronization purposes.

Each edge e in G_{ipc} represents the *synchronization constraint*

$$start(snk(e), k) \geq end(src(e), k - del(e)) \text{ for all } k, \tag{48}$$

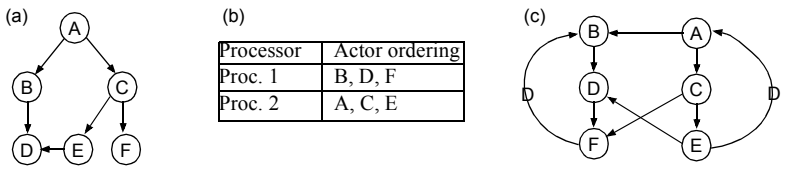

Figure 8. An illustration of a self-timed schedule and its associated IPC graph.

where $start(v, k)$ and $end(v, k)$ respectively represent the times at which firing k of actor v begins execution and completes execution.

Initially, the synchronization graph G_s is identical to G_{ipc}. However, various transformations can be applied to G_s in order to make the overall synchronization structure more efficient. After all transformations on G_s are complete, G_s and G_{ipc} can be used to map the given parallel schedule into an implementation on the target architecture. The IPC edges in G_{ipc} represent buffer activity, and are implemented as buffers in shared memory, whereas the synchronization edges of G_s represent synchronization constraints, and are implemented by updating and testing flags in shared memory. If there is an IPC edge as well as a synchronization edge between the same pair of actors, then a synchronization protocol is executed before the buffer corresponding to the IPC edge is accessed to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two actors in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two actors but no IPC edge, then no shared buffer is allocated between the two actors; only the corresponding synchronization protocol is invoked.

Any transformation that we perform on the synchronization graph must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), we say that G_1 *preserves* G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have $\rho_{G_1}(src(e), snk(e)) \leq del(e)$, where $\rho_G(x, y) \equiv \infty$ if there is no path from x to y in the synchronization graph G, and if there is a path from x to y, then $\rho_G(x, y)$ is the minimum over all paths p directed from x to y of the sum of the edge delays on p. The following theorem (developed in [9]) underlies the validity of a variety of useful synchronization graph transformations, which we discuss in Sections 5.1-5.4.

Theorem 1 The synchronization constraints (as specified by (48)) of G_1 imply the constraints of G_2 if G_1 preserves G_2.

5.1 Removal of redundant synchronization edges

A synchronization edge is *redundant* in a synchronization graph G if its removal yields a graph that preserves G. Equivalently, a synchronization edge e is redundant if there is a path $p \neq (e)$ from $src(e)$ to $snk(e)$ such that $\delta(p) \leq del(e)$, where $\delta(p)$ is the sum of the edge delays on path p. Thus, the synchronization function associated with a redundant synchronization edge "comes for free" as a by product of other synchronizations.

Example 8: Figure 9 shows an example of a redundant synchronization edge. The dashed edges in this figure are synchronization edges. Here, before executing actor D, the processor that executes $\{A, B, C, D\}$ does not need to synchronize with the processor that executes $\{E, F, G, H\}$ because due to the synchronization edge x_1, the corresponding firing of F is guaranteed to complete before each firing of D is begun. Thus, x_2 is redundant.

The following result establishes that the order in which we remove redundant synchronization edges is not important.

Theorem 2 [9] Suppose $G_s = (V, E)$ is a synchronization graph, e_1 and e_2 are distinct redundant synchronization edges in G_s, and $G_s = (V, E - \{e_1\})$. Then e_2 is redundant in G_s.

Theorem 2 tells us that we can avoid implementing synchronization for *all* redundant synchronization edges since the "redundancies" are not interdependent. Thus, an optimal removal of redundant synchronizations can be obtained by applying a straightforward algorithm that successively tests the synchronization edges for redundancy in some arbitrary sequence, and removes each of the edges that are found to be redundant. Such testing and removal of redundant edges can be performed in $O(|V|^2 \log_2(|E|) + |V||E|)$ time.

Example 9: Figure 10(a) shows a synchronization graph that arises from a two-processor schedule for a four-channel multi-resolution QMF filter bank, which has applications in signal compression. As in Figure 9, the dashed edges are synchronization edges. If we apply redundant synchronization removal to the synchronization graph of Figure 10(a), we obtain the synchronization graph in Figure 10(b): the edges (A_1, B_2), (A_3, B_1), (A_4, B_1), (B_2, E_1), and (B_1, E_2) are detected to be redundant, and the number of synchronization edges is reduced from 8 to 3 as a result.

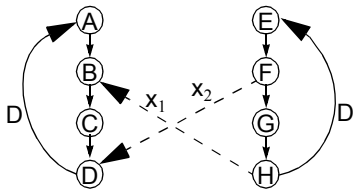

Figure 9. An example of a redundant synchronization edge.

5.2 Resynchronization

The goal of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus, the net synchronization cost is reduced. To ensure that the serialization introduced by resynchronization does not degrade the throughput, the new synchronizations are restricted to lie outside the SCCs of the synchronization graph (*feedforward resynchronization*) [49].

Resynchronization of self-timed multiprocessors has been studied in two contexts [10]. In *maximum-throughput resynchronization*, the objective is to compute a resynchronization that minimizes the total number of synchronization edges over all synchronization graphs that preserve the original synchronization graph. It has been shown that optimal resynchronization is NP-complete. However, a broad class of synchronization graphs has been identified for which optimal resynchronization can be performed by an efficient, polynomial-time algorithm. A heuristic for general synchronization graphs called Algorithm *Global-resynchronize* has also been developed that works well in practice.

Effective resynchronization improves the throughput of a multiprocessor implementation by reducing the rate at which synchronization operations must be performed. However, since additional serialization is imposed by the new syn-

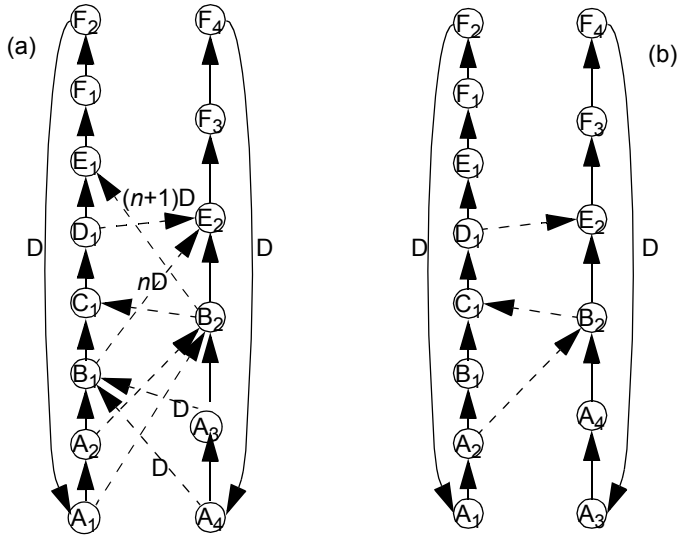

Figure 10. An example of redundant synchronization removal.

chronizations, resynchronization can produce a significant increase in latency. In *latency-constrained resynchronization*, the objective is to compute a resynchronization that minimizes the number of synchronization edges over all valid resynchronizations that do not increase the latency beyond a pre-specified upper bound on the tolerable latency. Latency-constrained resynchronization is intractable even for the very restricted sub-class of synchronization graphs in which each SCC contains only one actor, and all synchronization edges have zero delay. However, an algorithm has been developed that computes optimal latency-constrained resynchronizations for two-processor systems in $O(N^2)$ time, where N is the number of actors. Also, an efficient extension of Algorithm *Global-resynchronize,* called Algorithm *Global-LCR*, has been developed for latency-constrained resynchronization of general synchronization graphs.

Figure 11 illustrates the results delivered by *Global-LCR* when it is applied to a six-processor schedule of a synthesizer for plucked-string musical instruments in 11 voices. The plot in Figure 11 shows how the number of syn-

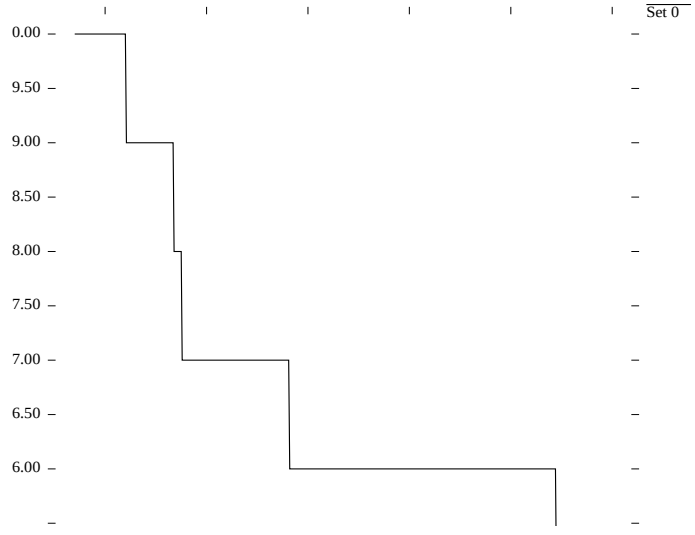

Figure 11. An illustration of resynchronization.

chronization edges in the result computed by *Global-LCR* changes as the latency constraint varies. The alternative synchronization graphs represented in Figure 11 offer a variety of latency/throughput trade-off alternatives for implementing the given schedule. The (right-most) extreme of these trade-off points offers 22% to 27% improvement in throughput, and 32% to 37% reduction in the average rate at which shared memory is accessed, depending on the access time of the shared memory. Since accesses to shared memory typically require significant amounts of energy, this reduction in the average rate of shared memory accesses is especially useful when low power consumption is an important implementation issue.

5.3 Feedforward and feedback synchronization

In general, self-timed execution of a multiprocessor schedule can result in unbounded data accumulation on one or more more IPC edges. However, the following result states that each *feedback edge* (an edge that is contained in an SCC) has a bounded buffering requirement. This result emerges from the theory of *timed marked graphs*, a family of computation structures to which synchronization graphs belong.

Theorem 3 Throughout the self-timed execution of an IPC graph G_{ipc}, the number of tokens on a feedback edge e of G_{ipc} is bounded; an upper bound is given by

$$min(\{\delta(C) | (e \in edges(C))\}), \tag{49}$$

where $\delta(C)$ denotes the sum of the edge delays in cycle C. The constant bound specified by (49) is called the *self-timed buffer bound* of that edge.

A *feedforward edge* (an edge that is not contained in an SCC), however, has no such bound on the buffer size.

Based on Theorem 3, two efficient protocols can be derived for the implementation of synchronization edges. Given an IPC graph (V, E), and an IPC edge $e \in E$, if e is a feedforward edge then we can apply a synchronization protocol called *unbounded buffer synchronization* (*UBS*), which guarantees that $snk(e)$ never attempts to read data from an empty buffer (to prevent underflow), and $src(e)$ never attempts to write data into the buffer unless the number of tokens already in the buffer is less than some pre-specified limit, which is the amount of memory allocated to that buffer (to prevent overflow). If e is a feedback edge, then we use a simpler protocol, called *bounded buffer synchronization* (*BBS*), that only explicitly ensures that overflow does not occur. The simpler BBS protocol requires only half of the run-time overhead that is incurred by UBS.

5.4 Implementation using only feedback synchronization

One alternative to implementing UBS for a feedforward edge e is to add synchronization edges to G_s so that e becomes encapsulated in an SCC, and then implement e using BBS, which has lower cost. An efficient algorithm, called *Convert-to-SC-graph*, has been developed to perform this graph transformation in such a way that the net synchronization cost is minimized, and the impact on the self-timed buffer bounds of the IPC edges is optimized. *Convert-to-SC-graph* effectively "chains together" the source SCCs, chains together the sink SCCs, and then connects the first SCC of the "source chain" to the last SCC of the sink chain with an edge. Depending on the structure of the original synchronization graph, *Convert-to-SC-graph* can reduce the overall synchronization cost by up to 50%.

Since conversion to a strongly connected graph must introduce one or more new cycles, it may be necessary to insert delays on the edges added by *Convert-to-SC-graph*. These delays may be needed to avoid deadlock and to ensure that the serialization introduced by the new edges does not degrade the throughput. The location (edge) and magnitude of the delays that we add are significant since (from Theorem 3) they affect the self-timed buffer bounds of the IPC edges, which in turn determine the amount of memory that we allocate for the corresponding buffers.

A systematic technique has been developed, called Algorithm *Determine Delays*, that efficiently inserts delays on the new edges introduced during the conversion to a strongly connected synchronization graph. For a broad class of practical synchronization graphs — those synchronization graphs that contain only one source SCC *or* only one sink SCC — Determine Delays computes a solution (placement of delays) that minimizes the sum of the resulting self-timed buffer bounds. For general synchronization graphs, Determine Delays serves as an efficient heuristic.

6 Block processing

Recall from Section 2.2 that DSP applications are characterized by groups of operations that are applied repetitively on large, often unbounded, data streams. *Block processing* refers to the uninterrupted repetition of the same operation (e.g., dataflow graph actor) on two or more successive elements from the same data stream. The *scalable synchronous dataflow* (*SSDF*) model is an extension of SDF that enables software synthesis of *vectorized* implementations, which exploit the opportunities for efficient block processing, and thus, form an important component of the cosynthesis design space. The internal specification of an SSDF actor A assumes that the actor will be executed in groups of $N_v(A)$ successive firings, which operate on $(N_v(A) \times cns(e))$-unit blocks of data at a

time from each incoming edge e. Block processing with well-designed SSDF actors reduces the rate of inter-actor context switching, and context switching between successive code segments within complex actors, and it also may improve execution efficiency significantly on deeply pipelined architectures.

At the Aachen University of Technology, as part of the COSSAP [44] software synthesis environment for DSP (now developed by Synopsys), Ritz, Pankert, and Meyr have investigated the optimized compilation of SSDF specifications [45]. This work has targeted the minimization of the context-switch overhead, or the average rate at which *actor activations* occur. An actor activation occurs whenever two distinct actors are invoked in succession. Activation overhead includes saving the contents of registers that are used by the next actor to invoke, if necessary, and loading state variables and buffer pointers into registers.

For example, the schedule

$$(2(2B)(5A))(5C) \tag{50}$$

results in five activations per schedule period. Parenthesized terms in (50) represent *schedule loops*, which are repetitive firing patterns that are to be translated into loops in the target code. More precisely, a parenthesized term of the form $(nT_1T_2\ldots T_n)$ specifies the successive repetition n times of the subschedule $T_1T_2\ldots T_n$. Schedules that contain only one appearance of each actor, such as the schedule of (50), are referred to as *single appearance schedules*. Because of their code size optimality, and because they have been shown to satisfy a number of useful formal properties [11], single appearance schedules have been the focus of a significant component of work in DSP software synthesis.

Ritz estimates the average rate of activations for a valid schedule S as the number of activations that occur in one iteration of S divided by the blocking factor $J(S)$. This quantity is denoted by $N'_{\text{act}}(S)$. For example, suppose we have an SDF graph for which $\mathbf{q}(A, B, C) = (10, 4, 5)$. Then

$$N'_{\text{act}}((2(2B)(5A))(5C)) = 5, \text{ and}$$
$$N'_{\text{act}}((4(2B)(5A))(10C)) = 9/2 = 4.5. \tag{51}$$

If for each actor, each firing takes the same amount of time, and if we ignore the time spent on computation that is not directly associated with actor firings (for example, schedule loops), then $N'_{\text{act}}(S)$ is directly proportional to the number of actor activations per unit time. In practice, these assumptions are seldom valid; however, $N'_{\text{act}}(S)$ gives a useful estimate and means for comparing schedules. For consistent acyclic SDF graphs, clearly N'_{act} can be made arbitrarily small by increasing the blocking factor sufficiently; thus, the extent to which the activation rate can be minimized is limited by the SCCs.

Ritz's algorithm for vectorization, which we call *complete hierarchization vectorization* (*CHV*), attempts to find a valid single appearance schedule that minimizes N'_{act} over all valid single appearance schedules. Minimizing the number of activations does not imply minimizing the number of appearances, and thus, the primary objective of CHV is, implicitly, code size minimization. As a simple example, consider the SDF graph in Figure 12. It can be verified that for this graph, the lowest value of N'_{act} that is obtainable by a valid single appearance schedule is 0.75, and one valid single appearance schedule that achieves this minimum rate is $(4B)(4A)(4C)$. However, valid schedules exist that are not single appearance schedules, and that have values of N'_{act} below 0.75; for example, the valid schedule $(4B)(4A)(3B)(3A)(7C)$ contains two appearances of A and B, and satisfies $N'_{act} = 5/7 = 0.71$.

In the CHV approach, the *relative vectorization degree* of a simple cycle C in a consistent, connected SDF graph $G = (V, E)$ is defined by

$$N_G(C) \equiv max(\{min(\{D_G(\alpha') \mid \alpha' \in parallel(\alpha)\}) \mid \alpha \in edges(C)\}), \quad (52)$$

where

$$D_G(\alpha) \equiv \left\lfloor \frac{del(\alpha)}{TNSE_G(\alpha)} \right\rfloor \quad (53)$$

is the delay on edge α normalized by the total number of tokens consumed by $snk(\alpha)$ in a minimal schedule period of G, and

$$parallel(\alpha) \equiv \{\alpha' \in E \mid src(\alpha') = src(\alpha) \textbf{ and } snk(\alpha') = snk(\alpha)\} \quad (54)$$

is the set of edges with the same source and sink as α.

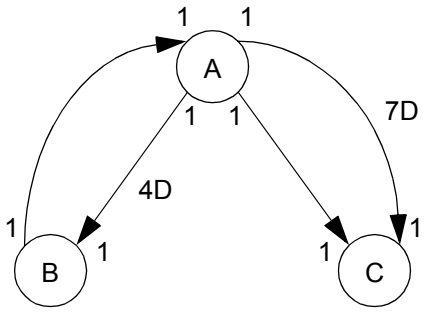

Figure 12. This example illustrates that minimizing actor activations does not imply minimizing actor appearances.

For example, if G denotes the graph in Figure 12 and χ denotes the cycle whose associated *vertices* set contains A and C, then $D_G(\chi) = \lfloor 7/1 \rfloor = 7$.

Given a strongly connected SDF graph, a valid single appearance schedule that minimizes N'_{act} can be constructed from a *complete hierarchization*, which is a cluster hierarchy such that only connected subgraphs are clustered, all cycles at a given level of the hierarchy have the same relative vectorization degree, and cycles in higher levels of the hierarchy have strictly higher relative vectorization degrees than cycles in lower levels [45].

Example 10: Figure 13 depicts a complete hierarchization of an SDF graph. Figure 13(a) shows the original SDF graph; here, $\mathbf{q}(A, B, C, D) = (1, 2, 4, 8)$. Figure 13(b), shows the top level of the cluster hierarchy. The hierarchical actor Ω_1 represents $subgraph(\{B, C, D\})$, and this subgraph is decomposed as shown in Figure 13(c), which gives the next level of the cluster hierarchy. Finally, Figure 13(d), shows that $subgraph(\{C, D\})$ corresponds to Ω_2 and is the bottom level of the cluster hierarchy. Now observe that the relative vectorization degree of the simple cycle in Figure 13(c) with respect to the original SDF graph is $\lfloor 16/8 \rfloor = 2$, while the relative vectorization degree of the simple cycle in Fig-

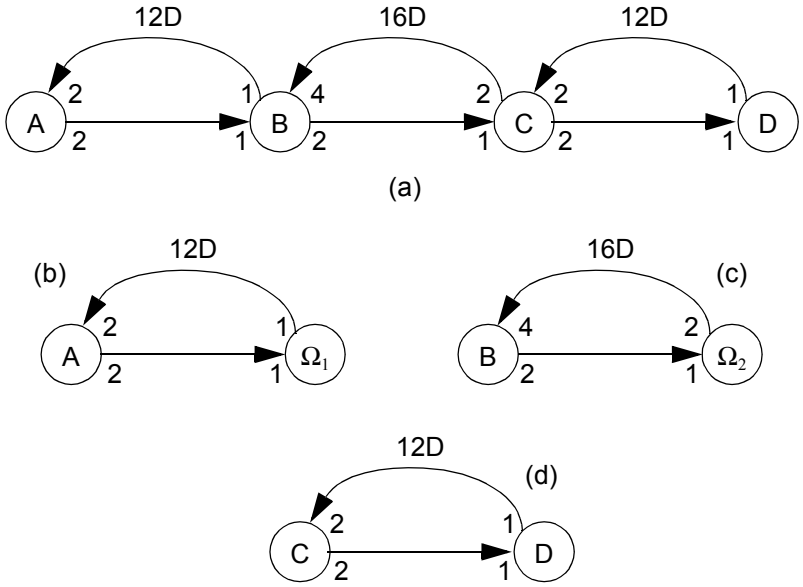

Figure 13. A complete hierarchization of a strongly connected SDF graph.

ure 13(b) is $\lfloor 12/2 \rfloor = 6$; and the relative vectorization degree of the simple cycle in Figure 13(d) is $\lfloor 12/8 \rfloor = 1$. Thus, we see that the relative vectorization degree decreases as we descend the hierarchy, and thus, the hierarchization depicted in Figure 13 is complete.

The hierarchization step defined by each of the SDF graphs in Figures 13(b)-(d) is called a *component* of the overall hierarchization.

The CHV technique constructs a complete hierarchization by first evaluating the relative vectorization degree of each simple cycle, determining the maximum vectorization degree, and then clustering the graphs associated with the simple cycles that do not achieve the maximum vectorization degree. This process is then repeated recursively on each of the clusters until no new clusters are produced. In general, this bottom-up construction process has unmanageable complexity; however, this normally does not create problems in practice since the SCCs of useful signal processing systems are often small, particularly in large grain descriptions.

Once a complete hierarchization is constructed, CHV constructs a schedule "template" — a sequence of loops whose iteration counts are to be determined later. For a given component Π of the hierarchization, if v_Π is the vectorization degree associated with Π, then all simple cycles in Π contain at least one edge α for which $D_G(\alpha) = v_\Pi$. Thus, if we remove from Π all edges in the set $\{\alpha | D_G(\alpha) = v_\Pi\}$, the resulting graph is acyclic, and if $F_{\Pi,1}, F_{\Pi,2}, \ldots, F_{\Pi,n_\Pi}$ is a topological sort of this acyclic graph, then valid schedules exist for Π that are of the form

$$T_\Pi \equiv (i_\Pi(i_{\Pi,1}F_{\Pi,1})(i_{\Pi,2}F_{\Pi,2})\ldots(i_{\Pi,n_\Pi}F_{\Pi,n_\Pi})) . \tag{55}$$

This is the subschedule template for Π.

Here, each $F_{\Pi,j}$ is a vertex in the hierarchical SDF graph G_Π associated with Π. Thus, each $F_{\Pi,j}$ is either a *base block* — an actor in the original SDF graph G — or a hierarchical actor that represents the execution of a valid schedule for the corresponding subgraph of G. Now let A_Π denote the set of actors in G that are contained in G_Π and in all hierarchical subgraphs nested within G_Π; and let $k_\Pi \equiv gcd(\{i_{\Pi,j} | 1 \le j \le n_\Pi\})$. Thus, we have

$$i_{\Pi,j} = k_\Pi \mathbf{q}_{G_\Pi}(F_{\Pi,j}), j = 1, 2, \ldots, n_\Pi . \tag{56}$$

The number of activations that T_Π contributes to N'_{act} is given by $((|B_\Pi| q_G(A_\Pi))/k_\Pi)$, where B_Π is the set of base blocks in G_Π [45]. Thus, if H denotes the set of hierarchical components in the given complete hierarchization, then

$$N'_{\text{act}} = \sum_{\Pi \in H} \frac{|B_\Pi| q_G(A_\Pi)}{k_\Pi}. \quad (57)$$

In the CHV approach, an exhaustive search over all i_Π and k_Π is carried out to minimize (57). The search is restricted by constraints derived from the requirement that the resulting schedule for G be valid. As with the construction of complete hierarchizations, it is argued that the simplicity of SCCs in most practical applications permits this expensive evaluation scheme.

Joint optimization of vectorization and buffer memory cost is developed in [46], and adaptations of the retiming transformation to improve vectorization for SDF graphs is addressed in [29, 55].

7 Summary

In this chapter, we have reviewed techniques for mapping high-level specifications of DSP applications into efficient hardware/software implementations. Such techniques are of growing importance in DSP design technology due to the increased use of heterogeneous multiprocessor architectures in which processing components, such as the ones discussed in Chapters 1-5, incorporate varying degrees and forms of programmability. We have discussed specification models based on coarse-grain dataflow principles that expose valuable application structure during cosynthesis. We then developed a number of systematic techniques for partitioning coarse-grain dataflow specifications into the hardware and software components of heterogeneous architectures for embedded multiprocessing. Synchronization between distinct processing elements in a partitioned specification was then discussed, and in this context, we examined a number of complementary strategies for reducing the execution-time and power consumption penalties associated with synchronization. We also reviewed techniques for effectively incorporating block processing optimization into the software component of a hardware/software implementation to improve system throughput.

Given the vast design spaces in hardware/software implementation, and the complex range of design metrics (e.g., latency, throughput, peak and average power consumption, memory requirements, memory partitioning efficiency, and overall dollar cost), important areas for further research include developing and precisely characterizing a better understanding of the interactions between different implementation metrics during cosynthesis; of relationships between various classes of architectures and the predictability and efficiency of implementations with respect to different implementation metrics; and of more powerful modeling techniques that expose additional application structure in innovative ways, and handle dynamic application behavior (such as the dynamic dataflow models and dataflow meta-models mentioned in Section 2.3). We expect all three of these

directions to be highly active areas of research in the coming years.

8 References

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets," in *Proceedings of the Design Automation Conference*, 1997, pp. 64–69.

[2] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Buffer Memory Requirements in DSP Applications," in *Proceedings of the International Workshop on Rapid System Prototyping*, 1994, pp. 198–123.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*: Addison-Wesley, 1988.

[4] A. L. Ambler, M. M. Burnett, and B. A. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms," *IEEE Computer Magazine*, vol. 25, 1992.

[5] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary Computation: Comments on the History and Current State," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 3–17, 1997.

[6] F. Balarin, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*: Kluwer Academic Publishers, 1997.

[7] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized Dataflow Modeling of DSP Systems," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, June, 2000, Istanbul, Turkey, To appear.

[8] B. Bhattacharya and S. S. Bhattacharyya. "Quasi-static scheduling of re-configurable dataflow graphs for DSP systems." *Proceedings of the International Workshop on Rapid System Prototyping*, Paris, France, June 2000. To appear.

[9] S. S. Bhattacharyya, S.Sriram, and E. A. Lee, "Optimizing Synchronization in Multiprocessor DSP Systems," *IEEE Transactions on Signal Processing*, vol. 45, 1997.

[10] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Resynchronization For Multiprocessor DSP Systems," *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, To appear.

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

[12] G.Bilsen, M.Engels, R.Lauwereins, and J. A. Peperstraete, "Cyclo-Static Dataflow," *IEEE Transactions on Signal Processing*, vol. 44, pp. 397–408, 1996.

[13] T. Blickle, J. Teich, and L. Thiele, "System-level Synthesis using Evolutionary Algorithms," *Journal of Design Automation for Embedded Systems*, pp. 23–58, 1998.

[14] J. T. Buck, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-Valued Control Systems," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 1994.

[15] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs Using the Token Flow Model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1993.

[16] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. "Ptolemy: A framework for simulating and prototyping heterogeneous systems." *International Journal of Computer Simulation*, January 1994.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[18] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-Software Co-Synthesis of Embedded Systems," in *Proceedings of the Design Automation Conference*, 1997.

[19] S. M. H. De Groot, S.Gerez, and O.Herrmann, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, pp. 351–364, May, 1992.

[20] G. De Micheli, *Synthesis and Optimization of Digital Circuits*: McGraw-Hill, 1994.

[21] G. De Micheli and M. Sami, *Hardware-software Co-design*: Kluwer Academic Publishers, 1996.

[22] R. Ernst, J. Henkel, and T. Benner, "Hardware-software Cosynthesis for Microcontrollers," *IEEE Design and Test of Computers Magazine*, pp. 64–75, 1993.

[23] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions On Computer-aided Design of Integrated Circuits And Systems*, Vol. 18, No. 6, June 1999.

[24] P. Hoang and J. Rabaey, "Hierarchical Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput," in *Proceedings of the International Conference on Application Specific Array Processors*, 1992.

[25] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, 1961.

[26] B. Jacob, "Hardware/Software Architectures for Real-Time Caching," in *Proceedings of the International Workshop on Compiler and Architecture Support for Embedded Systems*, October, 1999.

[27] A. Kalavade and E. A. Lee, "A Global Critically/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem," in *Proceedings of the International Workshop on Hardware/Software Co-Design*, 1994, pp. 42–48.

[28] A. Kalavade and P. A. Subrahmanyam, "Hardware/Software Partitioning for Multifunction Systems," *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 819–837, 1998.

[29] K. N. Lalgudi, M. C. Papaefthymiou, and M.Potkonjak, "Optimizing Systems for Effective Block-Processing: The k-Delay Problem," in *Proceedings of the Design Automation Conference*, 1996, pp. 714–719.

[30] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "GRAPE-II: A System-Level Prototyping Environment for DSP Applications," *IEEE Computer Magazine*, pp. 35–43, 1995.

[31] E. A. Lee and S.Ha, "Scheduling Strategies for multiprocessor real time DSP," in *Global Telecommunications Conference*, 1989.

[32] E. A. Lee, W. H. Ho, E.Goei, J.Bier, and S. S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, 1989.

[33] E. A. Lee, "Embedded Software — An Agenda for Research," Technical report. Electronics Research Laboratory, University of California at Berkeley UCB/ERL M99/63, December 1999.

[34] E. A. Lee and D. G. Messerschmitt, "Synchronous Dataflow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[35] E. A. Lee, "Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1993, pp. 453–456.

[36] Y.-T. S. Li and S.Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 1477–1487, 1997.

[37] Y. Li and W. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," in *Proceedings of the International Conference on Computer-Aided Design*, 1998, pp. 430–436.

[38] K. J. R. Liu, A. Wu, A. Raghupathy, and J. Chen, "Algorithm-Based Low-Power and High-Performance Multimedia Signal Processing," *Proceedings of the IEEE*, vol. 86, pp. 1155–1202, 1998.

[39] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*: Kluwer Academic Publishers, 1995.

[40] D. R. O'Hallaron, "The ASSIGN Parallel Program Generator," Technical report. School of Computer Science, Carnegie Mellon University May 1991.

[41] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1994.

[42] T. M. Parks, J. L. Pino, and E. A. Lee, "A Comparison of Synchronous and Cyclo-Static Dataflow," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 1995.

[43] P. Paulin, C. Liem, T. May, S. Sutarwala, "DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective," *Journal of VLSI Signal Processing*, pp. 23-47, January, 1995.

[44] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," in *Proceedings of the International Conference on Application Specific Array Processors*, 1992.

[45] 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*, 1993.

[46] S. Ritz, M. Willems, and H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1995.

[47] G. C. Sih, "Multiprocessor Scheduling to account for Interprocessor Communication,", Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1991.

[48] S. Sriram and E. A. Lee, "Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors," *Journal of VLSI Signal Processing*, pp. 207–220, 1997.

[49] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*: Marcel Dekker, Inc., 2000.

[50] D. E. Thomas, J. K. Adams, and H. Schmitt, "A Model and Methodology for Hardware/Software Codesign," *IEEE Design and Test of Computers Magazine*, vol. 10, pp. 6–15, 1993.

[51] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*: Prentice Hall, 1993.

[52] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[53] S. Wuytack, J.-P. Diguet, F. V. M. Catthoor, and H. J. De Man, "Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory Mappings," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, pp. 529–537, 1998.

[54] T.-Y. Yen and W. Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 1125–1136, 1998.

[55] V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processin*g, April 1994.