Parameterized Looped Schedules for Compact Representation of Execution Sequences in DSP Hardware and Software Implementation

Ming-Yung Ko, Claudiu Zissulescu, Sebastian Puthenpurayil, Shuvra S. Bhattacharyya, *Member, IEEE*, Bart Kienhuis, and Ed F. Deprettere, *Fellow, IEEE*

***Abstract*—In this paper, we present a technique for compact representation of execution sequences in terms of efficient looping constructs. Here, by a looping construct, we mean a compact way of specifying a finite repetition of a set of execution primitives. Such compaction, which can be viewed as a form of hierarchical run-length encoding (RLE), has application in many very large scale integration (VLSI) signal processing contexts, including efficient control generation for Kahn processes on field-programmable gate arrays (FPGAs), and software synthesis for static dataflow models of computation. In this paper, we significantly generalize previous models for loop-based code compaction of digital signal processing (DSP) programs to yield a configurable code compression methodology that exhibits a broad range of achievable tradeoffs. Specifically, we formally develop and apply to DSP hardware and software synthesis a parameterizable loop scheduling approach with compact format, dynamic reconfigurability, and low-overhead decompression.**

***Index Terms*—Design automation, embedded systems, field-programmable gate arrays (FPGAs), high-level synthesis, program compilers, reconfigurable design, signal processing.**

I. Introduction

DUE to tight resource constraints, strict real-time performance requirement, and the increasing complexity of applications, efficient program compression techniques are desired in the implementation of embedded digital signal processing (DSP) systems. Hardware and software subsystems for DSP often present periodic and deterministic execution sequences that facilitate compile- or synthesis-time compression. In this paper, we develop a methodology that exploits this characteristic of DSP subsystems through compact representation of execution sequences in terms of efficient looping constructs. The looping constructs provide a concise, parameterized way of specifying sequences of execution primitives that may exhibit repetitive patterns of arbitrary forms both at the primitive and subsequence levels. Such compaction provides a form of hierarchical run-length encoding (RLE) as well as reconfigurability during DSP system implementation. Moreover, exploitation of low-cost hardware features are considered to further improve the efficiency of the proposed methods. The power and flexibility of our approach is demonstrated concretely through its application to control generation for Kahn processes on field-programmable gate arrays (FPGAs) [9] and to software synthesis for static dataflow models of computation, such as those developed in [2], [16].

Manuscript received March 30, 2006; revised August 29, 2006. The associate editor coordinating the review of this paper and approving it for publication was Prof. Brian L. Evans.

M.-Y. Ko was with Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA. He is now with Sandbridge Technologies, Inc., White Plains, NY 10601 USA (e-mail: mko@sandbridgetech.com).

C. Zissulescu was with Leiden Institute of Advanced Computer Science, Leiden University, 2333 CA Leiden, The Netherlands. He is now with Chess BV, The Netherlands (e-mail: claudiu.zissulescu@chess.nl).

S. Puthepurayil is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: purayil@umd.edu).

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu).

B. Kienhuis and E. F. Deprettere are with the Leiden Institute of Advanced Computer Science, Leiden University, 2333 CA Leiden, The Netherlands (e-mail: kienhuis@liacs.nl; edd@liacs.nl).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSP.2007.893964

II. Related Work

Sequence compression, or *lossless compression* in the field of compression studies, techniques have been developed for many years in the context of file compression to save disk space, reduce network traffic, etc. One basic approach in this and other sequence compression domains is to express repeating strings of symbols in more compact forms. A typical example is RLE, which replaces repeated instances of a symbol by a single instance of the symbol along with the repetition count. Several bitmap file formats, e.g., TIFF, BMP, and PCX, adopt variants of RLE. More elaborate compression strategies include the dictionary-based approach (e.g., LZ77 [27]), prediction by partial matching through an adaptive statistical approach (PPM) [21], and block-sorting compression through the Burrows–Wheeler transform (BWT) [4].

Code compression in embedded systems presents some unique characteristics and challenges compared to compression in other domains. First, code sequences depend heavily on the underlying control flow structures of the associated programs. Furthermore, the control flow structures of the associated programs can often be changed subject to certain restrictions, giving rise in general to a family of alternative code sequences for the same program behavior. Second, memory resources in embedded systems are particularly limited, and the temporary "scratch space" for decompression is usually very small. Third,

decompression of embedded code must be fast enough to meet real-time demands.

Various research efforts are involved in the discussion of program size reduction. The work of [6] adopts classical compiler optimizations such as strength reduction, dead code elimination, and common subexpression elimination. A particularly effective strategy is procedural abstraction [19], where procedures are created to take the place of duplicated code sequences. The work of [5] further reveals that procedural abstraction combined with classical compiler optimizations result in more compact code size than each approach can achieve alone. Transparent program compression with little or no hardware support is proposed in [14] using a dictionary structure and procedure-based techniques. In [26], a scheme for block-based decompression in response to dynamic demands is presented to improve code density. A machine learning approach is proposed in [12] to compress programs through formalizing and automating decisions about instruction encoding that have traditionally been made by humans in a more ad hoc manner. An adaptive statistical technique (i.e., PPM-based) for code compression is presented in [10] that exploits the structure of program binaries to achieve superior compression ratios.

For embedded DSP design, application representations are often based on *data flow* models of computation, which exhibit certain advantages compared to traditional sequential programming. Data-flow-based DSP design usually operates at a high level of program abstraction, e.g., in terms of basic blocks, nested loop behaviors, and coarse-grain subroutines. Furthermore, the control flow at this abstraction level is often highly predictable. To reduce code size cost, repetitive execution patterns generated by this form of predictable control flow can be mapped to low-overhead looping constructs that are common in programmable digital signal processors, and are similarly easy to emulate in programmable- or custom-hardware designs. *Synchronous data flow (SDF)* is a specialized form of data flow that greatly facilitates static analysis for a broad class of DSP applications [18]. SDF and closely related programming models are widely used in commercial and research-oriented tools for simulation and implementation of DSP systems [2].

The work of [3] adopts a dynamic programming approach to reformat repeated dataflow executions in a hierarchical RLE style. However, the computational complexity is relatively large, especially in hardware and software synthesis contexts. In [2], two complementary loop scheduling algorithms for data-flow-based DSP programs are proposed for joint code and data memory minimization. In the methods of [2] and [3], the constraint of static and fixed iteration counts in the targeted class of looping structures significantly restricts compression results. In [1], a metamodeling approach is developed for incorporating dynamic reconfiguration capability into different data flow modeling styles. When applied to SDF, this metamodeling framework results in the *parameterized synchronous data flow (PSDF)* model of computation. The developments in [1] center around a hybrid compile-time/run-time scheduling technique that is specialized to PSDF representations.

The *process network (PN)* model is also a popular computation model for DSP applications. The Compaan project applies PN as a high-level intermediate representation for applications written in Matlab [8]. The Compaan tool is restricted to operate on affine nested loop Matlab programs, which are commonly encountered in DSP algorithms. The work of [9] explores PN-based hardware synthesis for nested loop programs. Through PN modeling, loop iterations can be partitioned into polytopes for efficient analysis and derivation of parallel implementations.

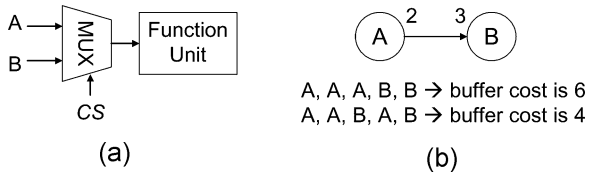


Fig. 1. Motivational examples: (a) Function unit with input data selected through a multiplexer. (b) SDF graph modeling for DSP software synthesis.

In this paper, we propose a flexible and parameterizable looping construct, and associated analysis methods. Because the approach is formulated in terms of compressing fixed execution sequences, this looping construct is applicable to any representation, such as SDF and *cyclo-static data flow (CSDF)* [16], from which static schedules can be derived. The looping construct provides compact format, dynamic reconfigurability, and fast decompression. The construct embeds functions in describing variable repetition lengths in a configurable form of RLE to achieve better compression results.

As a consequence, appropriate execution subsequences can be derived by adjusting parameter values at run time without modifying the hardware implementation. Our proposed methodology applies looping constructs that provide flexibility in adapting execution sequences, as well as efficiency in managing the associated iteration control. In summary, we propose an approach for compact representation of execution sequences that is effective across the dimensions of conciseness, decompression performance, cost, and configurability.

III. Motivation

Suppose we have a function unit (FU) with input data selected through a multiplexer (MUX) as in Fig. 1(a). The input data sources are A, B and the multiplexer has a control line (CS) for selecting one source at each instant. During the execution of FU, the multiplexer needs to determine a sequence of source executions to obtain proper input data. Such schematic structure is often implied in the hardware implementation of many DSP algorithms with intensive program loops, array element accesses, and matrix operations (more details are provided in Section VII).

One important issue is the implementation of such sequences. For example, we may have the following source selection sequence:

$$A,A,A,A,B,B,B,B,A,A,A,A,B,B,B,B,A,A,A,A,B,B,B,B.$$

On one hand, a small hardware area is favored for embedded systems and compact representation of sequences is useful for this objective. On the other hand, decoding logic must be efficient to meet real time requirements. Furthermore, dynamic pa-

rameterizability to generate multiple sequences is also desired, especially when a hardware implementation is fixed.

Another motivational example is from SDF-based software synthesis. In SDF modeling, execution sequences of subroutines have significant impact on data memory usage, as shown in Fig. 1(b). Wrapping periodically executed subroutine calls in loops is one way to reduce code size. However, optimal data memory usage often results from irregular execution patterns, and flexible looping strategies are needed to express such patterns. More details of this context are explored in Section VIII.

Therefore, we propose a compact representation of execution sequences, where decoding efficiency and parameterizability are both incorporated. Specifically, the compaction is done through identifying repeatedly occurring subsequences and reconstructing them in hierarchical, RLE-like structures for subsequence reuse.

In this paper, we introduce significant amounts of formal notation in certain places. Our introduction of formal notation is intended mainly for review of previous work on loop representations, precise discussion of a form of isomorphism that we apply in relating different loop representation instances to one another, and a technique that we apply for families of loop representations that are have affine relationships in their underlying parameters. The major notations used in this paper are summarized in an Appendix at the end of the paper.

IV. Static Looped Schedules

We denote the set of all integers by Z, and the set of non-negative integers by Z^+. Suppose $S = (s_1, s_2, \ldots, s_m)$ is a sequence of arbitrary elements and c is a non-negative integer. Then, we define the product $S \times c$ to be the sequence that results from concatenating c copies of S. Thus, for example $S \times 0$ is the empty sequence; $S \times 1 = S$; $S \times 2 = (s_1, s_2, \ldots, s_m, s_1, s_2, \ldots, s_m)$; and so on. Furthermore, if $T = (t_1, t_2, \ldots, t_n)$ is another sequence, then we define the sum $S + T$ to be the concatenation of T to S: $S + T = (s_1, s_2, \ldots, s_m, t_1, t_2, \ldots, t_n)$. Note that, in general, $S + T$ does not equal $T + S$.

Suppose we are given a finite sequence of symbols $P = (p_1, p_2, \ldots, p_n)$ from a finite alphabet set $A = \{a_1, a_2, \ldots, a_m\}$. Thus, each $p_i \in A$. We refer to each p_i as an *instruction*, and we refer to the sequence P as the *program* that we wish to optimize. We define a *class 0 (static) schedule loop* over A to be a parenthesized term of the form $(cI_1I_2 \ldots I_k)$, where $c \in Z^+$, and each I_i is either an element of A (i.e., an instruction) or a ("nested") class 0 schedule loop. The number c is called the *iteration count* of the schedule loop, and each I_i is called an *iterand* of the schedule loop. The concatenation $I_1I_2 \ldots I_k$ of iterands is called the *body* of the schedule loop. Such a schedule loop is called *static* because the iteration count is constant.

A *class 0 (static) looped schedule* over A is a sequence $S = (s_1, s_2, \ldots, s_n)$, where each s_i is either an element of A or a class 0 schedule loop over A. Note that by definition, if $L = (cI_1I_2 \ldots I_k)$ is a class 0 schedule loop, then $S_L = (I_1, I_2, \ldots, I_k)$ and (L) are both class 0 looped schedules. We call S_L the *body schedule* of L. When discussing looped schedules or schedule loops, "class 0" and "static" are equivalent.

Given a class 0 looped schedule S, a schedule loop L is *contained* in S if for some i, s_i is a schedule loop and $s_i = L$ or L is a schedule loop that is nested within s_i. For example, consider $S = ((3A(2B(3CD))), E, (3(2B)))$. This schedule contains the following schedule loops: $(3A(2B(3CD)))$, $(2B(3CD))$, $(3CD)$, $(3(2B))$, and $(2B)$. Note that in listing the set of schedule loops that are contained in a schedule, we may need to distinguish between multiple schedule loops that have identical iteration counts and bodies, as in the first and second appearances of $(3AB)$ in the looped schedule $((2A(3AB)), (5CD), (3AB))$. If L_1 and L_2 are schedule loops that are contained in the schedule $S = (s_1, s_2, \ldots, s_n)$, we say that L_1 is *contained earlier* than L_2 in S if there exist s_i and s_j such that $i < j$, s_i contains L_1, and s_j contains L_2. We say that L_1 *lexically precedes* L_2 in S if (a) L_1 is contained earlier than L_2 in S; (b) L_2 is nested within L_1; or (c) S contains a schedule loop L_3 so that L_1 is contained earlier than L_2 in the body schedule of L_3.

Example 1: Consider the looped schedule $((2A(3B)), C, D, (3A(2(3B)(2C))))$. Let L_1 denote the first appearance of $(3B)$, let L_2 denote the second appearance of $(3B)$, let L_3 denote the schedule loop $(2A(3B))$, and let L_4 denote the schedule loop $(2C)$. Then, L_1 lexically precedes L_2 due to condition a); L_3 lexically precedes L_1 due to condition b); and L_2 lexically precedes L_4 due to condition c).

Consider an iterand I of a class 0 schedule loop. If I is an instruction, then we say that the *program generated by* I, denoted $P(I)$, is simply the one-element sequence (I). Otherwise, if I is a schedule loop—that is, I is of the form $I = (cI_1I_2 \ldots I_n)$—then $P(I)$ is defined recursively by

$$P(I) = (P(I_1) + P(I_2) + \ldots + P(I_n)) \times c.$$

Similarly, given a class 0 schedule $S = (s_1, s_2, \ldots, s_n)$, the program generated by S is (with a minor abuse of notation) denoted $P(S)$ and is given by

$$P(S) = P(s_1) + P(s_2) + \ldots + P(s_n).$$

Example 2: Suppose that the set of instructions A is given by $A = \{a, b, c, d\}$, and suppose we are given a looped schedule $S = (a, (2c(2ad)d), b, (3c), d)$. Then, we have

$$P(S) = (a, c, a, d, a, d, d, c, a, d, a, d, d, b, c, c, c, d).$$

Static looped schedules have been studied extensively in the context of software synthesis from SDF representations of DSP applications (e.g., see [2]).

If costs are associated with individual actors and with loop construction in general, then we can express the degree of compactness associated with specific looped schedules. Suppose that in the context of looped schedule implementation, α_{loop} represents the overhead (cost) of a loop, and $\alpha(x)$ represents the cost of an instruction x. For example, for software implementation α_{loop} represents the code size cost associated with a loop in the target code. This value will normally depend on the processor on which the schedule is being implemented,

and will include the code size of the instructions required to initialize the loop and update the loop counter at the beginning or end of each iteration. If the software is being implemented for a dataflow graph specification, then the "instructions" in the looped schedule correspond to actors in the dataflow graph, and the instruction code size values $\{\alpha(x)\}$ give the code size requirements of the different actors on the associated target processor.

We occasionally abuse notation by overloading the definition of a function depending on the type of argument that is applied. For example, as explained fully in subsequent sections, if X is an instruction, then $\alpha(X)$ defines the cost of that instruction, whereas if X is a schedule, then $\alpha(X)$ denotes the total cost of that schedule (including the sum of instruction and loop costs). We abuse notation in this way to highlight relationships across closely related functions and to contain the total number of distinct symbols that are defined.

The cost of a looped schedule S can be expressed as

$$\alpha(S) = n_{\text{loop}}(S)\alpha_{\text{loop}} + \sum_{x \in A} n_{\text{app}}(x, S)\alpha(x)$$

where $n_{\text{loop}}(S)$ denotes the number of schedule loops in S (including nested loops), and $n_{\text{app}(x,S)}$ denotes the number of times that instruction x appears in schedule S. For example, if $S = (a, (2c(2ad)d), b, (3c), d)$, the schedule illustrated above, then

$$\alpha(S) = 3\alpha_{\text{loop}} + 2\alpha(a) + \alpha(b) + 2\alpha(c) + 3\alpha(d).$$

To construct a static looped schedule from a sequence of instructions, a dynamic programming algorithm called *CDPPO (dynamic programming post-optimization for code size minimization)* [3] provides an effective approach. The CDPPO algorithm adopts a bottom-up approach to fuse repetitive instruction sequences into hierarchical looping constructs. The objective of CDPPO is to minimize overall code size, including the costs for instructions and looping constructs. CDPPO has computational complexity that is polynomial in the number of instructions in the (uncompressed) input sequence.

V. Class 1 Looped Schedules

Static looped schedules provide a simple form of nested iteration where all iteration counts in the loops are static values, and loop counts implicitly progress from 1 to the corresponding iteration count limits in uniform steps of 1. However, static looped schedules do not always allow for the most compact representation of a static execution sequence. This motivates the definition of more flexible schedules in which more general updating of loop counters is integrated into the schedule. The class 1 schedules, which we define next, represent one such form of more general schedules. In class 1 schedules, the loop counter dimension is made explicit, and loop counters are allowed to have initial values, and update expressions specified for them. Because update expressions are processed frequently (once per loop iteration), their form is restricted in class 1 schedules to ensure efficient hardware and software implementation.

Formally, a class 1 schedule loop L has five attributes, a body, an index, an iteration count function, an initial index value, and an index update constant. The body of L is defined in a manner analogous to the body of a class 0 schedule loop. Thus, the body of L is of the form $I_1 I_2 \ldots I_n$, where each I_i, called an *iterand* of L, is either an instruction or a class 1 schedule loop. The *index* of a class 1 schedule loop L is a symbol that represents a loop index variable that is associated with L in an implementation of the loop. The *iteration count function* of L is an integer-valued function $f(y_1, y_2, \ldots, y_m)$ defined on Z^m, where each y_i is the index of some other class 1 schedule loop or is a parameter of a looped schedule that contains L. The value of f just before executing an invocation of L gives the minimum value of the index required for the loop to stop executing. In other words, L will continue executing as long as the index value is less than f. It is admissible to have $m = 0$, so that f represents a constant value. The *initial index value* of L is an integer to which the loop index variable associated with L is initialized. This initialization takes place before each invocation of L, just prior to the computation of f. The *index update constant* is a positive integer that is added to the index of L at the end of each iteration of L.

A class 1 schedule loop L is represented by the parenthesized term $([x_L, f_L, u_L]B_L)$, where x_L, f_L, B_L, and u_L are, respectively, the index, iteration count function, body, and index update constant of L. For brevity we omit the initial index value from this representation. The initial index value of L is denoted by $x_L(0)$; this value is specified separately when needed. Furthermore, when $u_L = 1$, we may suppress u_L from the schedule loop notation, and simply write $L = ([x_L, f_L]B_L)$. If x_L is not an argument of any relevant iteration count function, we may suppress x_L, and write $L = ([f_L]B_L)$; if, additionally, f_L is constant-valued (i.e., $f_L = c$), and $u_L = 1$, then we have a class 0 schedule loop, and we may drop the brackets and write $L = (cB_L)$, which is just the usual notation for class 0 schedule loops. We represent the arguments of the iteration count function by $args(f_L) = \{y_1, y_2, \ldots, y_m\}$. It is a fact that the number of iterations executed by an invocation of the class 1 schedule loop L is given by

$$\text{iterations} = \max\left(0, \left\lceil \frac{f_L(y_1^*, y_2^*, \ldots, y_m^*) - x_L(0)}{u_L} \right\rceil\right) \quad (1)$$

where y_i^* denotes the value of index y_i just prior to initiation of I_L.

A *class 1 looped schedule* over A is an ordered pair $S = (\text{params}(S), \text{body}(S))$. The first member $\text{params}(S) = \{p_1, p_2, \ldots, p_r\}$ of this ordered pair is a finite set of elements called *parameters* of S, and the second member $\text{body}(S) = (b_1, b_2, \ldots, b_n)$ is a finite sequence, where each b_i is either an element of A or a class 1 schedule loop over A.

Intuitively, the semantics of executing a class 1 schedule loop $([x_L, f_L, u_L]B_L)$, where $f_L = f_L(y_1, y_2, \ldots, y_m)$ and $u_L \in Z^+$, can be described as outlined in Fig. 2. Using this semantics, we can define the program generated by an iterand of a class 1 schedule loop, and the program generated by a class 1 looped schedule in a fashion analogous to the corresponding definitions for class 0 looped schedules. However, when determining these generated programs for class 1 looped schedules,

```
x_L = x_L(0)
limit_L = f_L(y_1, y_2, ..., y_m)
while (x_L < limit_L)
   execute B_L
   x_L = x_L + u_L
end while
```

Fig. 2. Sketch of the execution of a schedule loop.

we must specify an assignment of values to the schedule parameters. Thus, if $v : \text{params}(S) \rightarrow Z$ is an assignment of values to parameters of a class 1 looped schedule S, then we write $P(S, v)$ to represent the corresponding program generated by S.

Example 3: Provided an alphabet set $\{A, B, C, D, E, F\}$, and consider the class 1 looped schedule S specified by $\text{params}(S) = \{p_1\}$ and

$$\text{body}(S) = (F, ([x_1, f_1]AB([f_2, 2]CD)), E)$$

where $f_1 = f_1(p_1) = p_1 - 3$ and $f_2 = f_2(x_1) = 5 - x_1$. Notice that this schedule contains a pair of nested schedule loops. If the initial index values in these loops are identically zero, and if $v(p_1) = 6$ (i.e., we assign the value of 6 to the schedule parameter p_1), then we have

$$P(S, v) = (F, A, B, C, D, C, D, C, D, A, B, C, D, C, D, A, B, C, D, C, D, E).$$

This simple example illustrates some of the ways in which more irregular programs can be generated by class 1 looped schedules as compared to their class 0 counterparts. In particular, in this example, we see that the number of iterations of the inner loop can vary across different invocations of the loop, and furthermore, the amount of this variation need not be uniform.

Containment of a schedule loop earlier than another, as well as lexical precedence between schedule loops, are defined for class 1 looped schedules in a manner analogous to that for class 0 looped schedules.

We say that a looped schedule S is *syntactically correct* if the following three conditions all hold.

- Every loop $L = ([x_L, f_L, u_L]B_L)$ that is contained in S has a unique index x_L.
- $\{x_L | S \text{ contains } L\} \cap \text{params}(S) = \emptyset$; that is, the parameters of S are distinct from the loop indices.
- For each loop L that is contained in S, the iteration count function f_L is either constant-valued, or depends only on parameters of S, and indexes of loops that lexically precede L; that is,

$$\text{args}(f_L) \subseteq \text{params}(S) \cup \{x_{L'} | L' \text{ lexically precedes } L \text{ in } S\}.$$

Syntactic correctness is a necessary but not sufficient condition for validity of a looped schedule. Overall validity in general depends also on the context of the looped schedule. For example, a syntactically correct looped schedule for an SDF graph may be invalid because the schedule is deadlocked (attempts to execute an actor before sufficient data has been produced for it).

Because of their potential for parameterization, in terms of schedule parameters and loop indices, and because of their restriction that loop indexes be updated by constant additions, we also refer to class 1 looped schedules as *parameterized, constant-update looped schedules (PCLSs)*. Constant update of loop indexes is motivated primarily from efficiency considerations—most notably, from the constant update operations that are available for address registers in many DSP processors. The simplicity of constant updates also facilitates the derivation of efficient compression algorithms.

In the following section, the PCLSs that we consider are further restricted to those that use linear representations in specifying iteration count functions. This restriction is also related to considerations of efficiency in implementing PCLS structures, and in facilitating their derivation, especially in the compression of raw sequences (as opposed to compression from models that generate the sequences implicitly). Techniques based on these restrictions for automatically deriving PCLSs from raw sequences are then presented. In Section VII, an application domain where these approaches can be applied is demonstrated.

On the other hand, PCLSs are also useful when the sequence generation mechanism is known for specific models of computation. In Section VIII, we study this context of PCLS derivation. Here, PCLSs are used to generate SDF actor execution sequences that have low data memory cost. This kind of PCLS formulation requires full knowledge of the computation model (in this case, SDF), and is therefore oriented specifically to the application domain associated with the model. In other words, compaction of arbitrary nonlooped (raw) execution sequences is not applicable in this case.

VI. Affined Looped Schedules

One useful special case of PCLS arises when f_L is a *linear* function of $args(f_L)$. We call this special case *affine parameterized looped schedules (APLSs)*. Because of the linearity property, linear algebra theories can be used to develop computationally effective APLS formulation algorithms. Such algorithms are proposed in the following sub-sections and applied to compress execution sequences of an FPGA application domain in Section VII.

A. Isomorphism of Looped Schedules

The ability to parameterize iteration counts in PCLSs is useful in expressing related groups of static schedule loops. In many useful design contexts, families of static schedule loops arise, such that within a given family, all loops are equivalent in a certain structural sense. We refer to this form of equivalence between loops as schedule loop *isomorphism*. Specifically, two class 0 schedule loops L_1 and L_2 are isomorphic if there is a bijection θ between the set of loops contained in (L_1) and the set of loops contained in (L_2) such that for each L in the domain of θ, $L = (cI_1I_2 \ldots I_m)$ and $\theta(L) = (dJ_1J_2 \ldots J_n)$ satisfy the following three conditions: 1) L and $\theta(L)$ have the same number of iterands (that is, $m = n$); 2) for each i such that I_i is not a loop (i.e., it is a "primitive" iterand), we have $J_i = I_i$; and 3) for each i such that I_i is a loop, we have that J_i is also a loop, and furthermore, I_i and J_i are isomorphic.

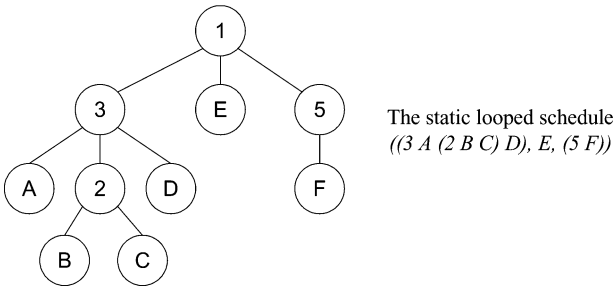


Fig. 3. Illustration of the generalized schedule tree (GST) representation.

For each loop L contained in L_1, the mapping $\theta(L)$ of L is called the *image* of L under the isomorphism. Furthermore, two static looped schedules S_1 and S_2 are said to be isomorphic if the loops $(1S_1)$ and $(1S_2)$ are isomorphic.

We can extend the definition of isomorphic looped schedules to a finite set of static looped schedules $S_1, S_2, \ldots, S_k$. In this case, we extract the loops from $(1S_i)$ for some arbitrary i. Then, for all $j \neq i$ and for each loop L contained in $1S_i$, we define $\theta_j(L)$ to be the corresponding, structurally equivalent loop in $(1S_j)$.

Determination of whether or not two looped schedules S_1 and S_2 are isomorphic can be performed in polynomial time by operating on tree-structured, graphical representations of these schedules, which we call *generalized schedule trees (GSTs)*. The GST representation generalizes to arbitrary looped schedules the *schedule tree* representation that was introduced in [22] (the schedule tree was defined for a restricted class of schedules called R-schedules). In a GST, each internal node Φ corresponds to a schedule loop L_Φ and is annotated with the iteration count of that schedule loop. The children of an internal node Φ correspond to the iterands of L_Φ, and these children are ordered according to the order of these iterands. From these construction rules, it follows that leaf nodes in the GST correspond to primitive iterands. An illustration of a GST is shown in Fig. 3.

Now, for $i = 1, 2$, let the number of schedule loops and primitive iterands contained in each S_i be denoted by $n_{L,i}$ and $n_{P,i}$, respectively. By constructing GSTs and comparing their nodes through parallel searches through the trees, we can determine whether or not two looped schedules are isomorphic in $O(max(n_{L,1}, n_{P,1}, n_{L,2}, n_{P,2}))$ time. Furthermore, once loops are found to be isomorphic, their pairs of corresponding looped schedules can be traversed efficiently by through the GST representation.

B. Basics of APLS Derivation

Using the concept of looped schedule isomorphism, we derive useful compaction formulations in this section for the special case of APLSs where $\text{args}(f_L) = \text{params}(S)$ for every L contained in S.

For clarity in this discussion, we start with p as the only schedule parameter (i.e., $\text{params}(S) = \{p\}$). Under the APLS assumption, this means that the iteration count expression for each loop will be of the form $ap+b$, where a and b are constants. Therefore, we need two instances of a given static schedule loop to fit the unknowns a and b. We simply need that these instances be for distinct values of p, say q_1 and q_2, and that these values of p be such that they reach beyond any transient effects (leading to negative, zero, or one-iteration schedule loops when viewed from the final parameterized schedule). Note that functionally, a negative-iteration schedule loop is just equivalent to a zero-iteration schedule loop.

Suppose now that we have an isomorphic schedule pair S_1 and S_2. (If S_1 and S_2 are not isomorphic, we need to increase $min(q_1, q_2)$, and try again.) We then take each schedule loop L in S_1 and its image $\theta(L)$ in S_2. Let z_1 be the iteration count of L and z_2 be that of $\theta(L)$. We then set up the equations

$$z_1 = aq_1 + b \text{ and } z_2 = aq_2 + b$$

and solve these equations for a and b. We repeat this procedure for all loops L that are contained in S_1.

Generalizing this to multiple schedule parameters, we start with a hypothesized APLS $S(p_1, p_2, \ldots, p_N)$ in $N \geq 1$ parameters. The iteration count expression for each schedule loop L is of the form $(a_1p_1 + a_2p_2 + \ldots + a_Np_N + b)$. We need $N+1$ instances of L to fit the $N+1$ unknowns in the iteration count expression for L. For $i = 1, 2, \ldots, N+1$, let S_i be the ith element in our set of compacted looped schedule instances. Let $q_{i,1}, q_{i,2}, \ldots, q_{i,N}$ be the corresponding parameter values for $p_1, p_2, \ldots, p_N$, respectively. Furthermore, let L be a schedule loop in S_i, and for each $i = 1, 2, \ldots, N+1$, let z_i denote the iteration count of $\theta_i(L)$. We set up the following equations:

$$\begin{aligned}
z_1 &= a_1q_{1,1} + a_2q_{1,2} + \cdots + a_Nq_{1,N} + b \\
z_2 &= a_1q_{2,1} + a_2q_{2,2} + \cdots + a_Nq_{2,N} + b \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
z_{N+1} &= a_1q_{N+1,1} + a_2q_{N+1,2} + \cdots + a_Nq_{N+1,N} + b
\end{aligned}$$

This can be expressed in matrix form as $\vec{z} = \mathbf{Q}\vec{a} + \vec{b}$, where $\vec{z}$ is an $(N+1) \times 1$ constant column vector, $\vec{a}$ is an $N \times 1$ column vector composed of the unknown a_i's, $\mathbf{Q}$ is an $(N+1) \times N$ constant matrix composed of the parameter settings used in the selected schedule instances, and $\vec{b} = [b\ b\ \ldots\ b]^T$ is an $(N+1) \times 1$ column vector obtained by replicating the unknown offset term b.

By solving the linear equations, we obtain $a_1, a_2, \ldots, a_N$ and b to formulate the APLS loop implementation of L. If a solution cannot be obtained, we can increase the selected $\{q_{i,1}, q_{i,2}, \ldots, q_{i,N}\}$ (to more completely bypass transient effects, as described earlier), or we may change the hypothesized number of parameters in the looped schedule.

Example 4: Given the system shown in Fig. 1, suppose that sequences are determined at run time by an integer parameter p, and that under parameter assignments $v_1(p) = 5$ and $v_2(p) = 6$, the corresponding sequences are $S_1 = (A, A, B, A, A, B)$ and

$$S_2 = (A, A, A, A, B, B, B, B, A, A, A, A, B, B, B, B, A, A, A, A, B, B, B, B)$$

respectively. These sequences can be compacted into the static schedule loops $L_1 = (2(2A)(1B))$ and $L_2 = (3(4A)(4B))$. Then, employing the APLS formulation techniques, the

two schedule loops can be unified into a singe expression $L = ([f_1]([f_2]A)([f_3]B))$, where $f_1(p) = p - 3$, $f_2(p) = 2p - 8$, and $f_3(p) = 3p - 14$.

C. Consolidating Loops Within a Schedule

While the previous subsection focuses on isomorphism across schedules, this subsection discusses isomorphism within a schedule. Let us first look at a class 0 looped schedule $S = ((1A), B, (2A), B, (3A), B)$. The schedule S cannot be compressed further by class 0 scheduling algorithms due to the heterogeneous iteration $(1A)$, $(2A)$, and $(3A)$. However, schedules $((1A), B)$, $((2A), B)$, and $((3A), B)$ contained in S are isomorphic to each other and our isomorphism-based compression technique is able to unify them in a single APLS loop. By inspection, we can easily evaluate this unified schedule loop to be $L = ([y_1, f_1]([f_2]A)B)$, where $f_1 = 3$, $f_2 = y_1 + 1$, and $y_1(0) = 0$.

Motivated by this example, we now describe a formal method to compute schedule loops of this kind in a general fashion. Given a static schedule $S = (s_1, s_2, \ldots, s_n)$, suppose that S contains z consecutive isomorphic subschedules and each subschedule contains w ($w \geq 1$) elements of S. Let the z consecutive subschedules be represented successively (from left to right) as

$$\begin{aligned} S_0 &= (s_i, s_{i+1}, \cdots, s_{i+w-1}) \\ S_1 &= (s_{i+w}, s_{i+w+1}, \cdots, s_{i+2w-1}) \\ &\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ S_{z-1} &= \left(s_{i+(z-1)w}, s_{i+(z-1)w+1}, \cdots, s_{i+zw-1}\right) \end{aligned}$$

where $i \geq 1$, and $i + zw - 1 \leq n$. In addition, given an integer j ($i \leq j \leq i + w - 1$), any pair of elements in the subset $\{s_j, s_{j+w}, \ldots, s_{j+(z-1)w}\}$ are isomorphic, and we therefore call this subset the *isomorphism family* ϕ_j. Furthermore, suppose that elements of each subschedule S_u ($0 \leq u \leq z - 1$)—i.e., $s_{i+uw}, s_{i+uw+1}, \ldots, s_{i+(u+1)w-1}$—are not uniformly isomorphic one another, and therefore, that $S_0, S_1, \ldots, S_{z-1}$ form an *isomorphism basis* (i.e., a decomposition into maximal isomorphic subschedules). Our goal is to consolidate these z subschedules into a single APLS schedule loop

$$L = \left([y, f_L]s'_i s'_{i+1} \ldots s'_{i+w-1}\right) \tag{2}$$

where $f_L() = z$ and s'_j is an iterand evaluated from the isomorphism family ϕ_j.

In our formulation, values of y are set as the subschedule subscripts, i.e., $y = t$ is for S_t. For any loop l contained in s'_j, we need to derive the loop iteration count function, say $f_l()$. For brevity, we discuss only the case of $args(f_l()) = \{y\}$; our treatment of this case can be extended however to more general case. After consolidating all the subschedules, the new S becomes $S = (S_L, L, S_R)$, where S_L (S_R) is a possibly-empty subschedule that immediately precedes S_0 (succeeds S_{z-1}) in the original schedule S.

Deriving L is then similar to that discussed in the previous subsection. The affine function of $f_l()$ is $ay + b$ and a, b are to be solved through the z isomorphic images in the isomorphism family ϕ_j.

D. Further Consolidation of Loops by Incorporating Schedule Parameters

The APLS derivation techniques in the previous subsection can be incorporated with schedule parameters for further compaction. Consider the following two static schedule instances that involve an associated parameter p:

$$\begin{aligned} S_{v(p)=5} &= (A, B, (2A), B, (3A), B) \text{ and} \\ S_{v(p)=6} &= ((3A), B, (4A), B, (5A), B, (6A), B)\,. \end{aligned}$$

By employing the basic APLS derivation technique, we obtain

$$\begin{aligned} S_{v(p)=5} &= (([x, f_1]\,([f_2]A)\,B)) \text{ and} \\ S_{v(p)=6} &= (([y, g_1]\,([g_2]A)\,B)) \end{aligned}$$

where $f_1 = 3$, $f_2 = x + 1$, $g_1 = 4$, and $g_2 = y + 3$. Since both APLSs are isomorphic to one another, there are chances to merge them into one. A new iteration count function $p - 2$ can replace both f_1 and g_1. For f_2 and g_2, the distinct constant shifts (i.e., 1 of $x + 1$ and 3 of $y + 3$) can be consolidated into one affine function $2p - 9$. Therefore, the compacted form is $S = (([i, h_1]([h_2]A)B))$, where $h_1 = p-2$ and $h_2 = i+2p-9$.

In generalizing this kind of derivation, we assume for simplicity here that we are working with schedules that involve a single parameter p only. Multiple parameters can be handled with a straightforward (but more notationally cumbersome) extension.

Suppose that schedule loop instances of (2) are provided. For any loop l contained in L, our goal is to relate the iteration count function $ay + b$ to p in affine functions. That is, $a = c'p + d'$ and $b = c''p + d''$, although this formulation may make ay a non-affine term. By solving for c', d', c'', and d'', we obtain the new iteration count function $(c'p + d')y + (c''p + d'')$ for l.

Example 5: Let us revisit the input selection example in Fig. 1(a). Suppose the selection sequences are generated by the pseudocode in Fig. 4(a) and p is an integer parameter. If the loop iteration space is drawn on a plane, it will look like Fig. 4(b) (with p set to 6). The sequence goes from bottom to top, starting at the left-most column, and traversing the columns from left to right. For example, the sequence for $v(p) = 6$ is

$$\begin{aligned} &(A, A, A, A, A, A, B, A, A, B, A, A, B, B, \\ &\qquad A, B, B, A, B, B, B, B, B, B). \end{aligned} \tag{3}$$

Our task now is to compact in APLS form the raw sequences associated with this example, assuming no prior knowledge about the sequence generation mechanism. To this end, the sequence in (3) can be first compacted as a static looped schedule, $((6A), (2B(2A)), (2(2B)A), (6B))$. The second and third elements are isomorphic and can be compacted further as $((6A), ([x, f_1](2([f_2]B)([f_3]A))), (6B))$, where $f_1 = 2$, $f_2 = x + 1$, and $f_3 = -x + 2$. Following the same procedure, we obtain a schedule for $v(p) = 7$, $((8A), ([y, g_1](2([g_2]B)([g_3]A))), (8B))$, where $g_1 = 3$, $g_2 = y + 1$, and $g_3 = -y + 3$. If p is incorporated, a unified schedule can be developed as

$$\left(([h_1]A), ([t, h_2]\,(2\,([h_3]B)\,([h_4]A))), ([h_5]B)\right)$$

(a)

```
for (int i = 1; i <= 2*p-4; i++) {
    for (int j = 4; j <= p; j++) {
        if ((j-4) < floor((i-1)/2)) {
            select B;
        } else {
            select A;
        }
    }
}
```

(b)

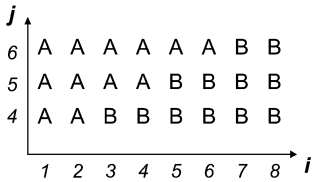


Fig. 4. (a) Pseudocode fragment that demonstrates the input selection sequence for Fig. 1. (b) Corresponding loop iteration space ($p = 6$).

where $h_1 = h_5 = 2p - 6$, $h_2 = p - 4$, $h_3 = t + 1$, and $h_4 = -t + p - 4$.

If transient effects are carefully considered, we can introduce further compression through certain forms of "dummy" iterands. For example, $(6A)$ can be rewritten as $(2(0B)(3A))$. Through this observation, the APLS can be reformulated as $(([t, h_1](2([h_2]B)([h_3]A))))$, where $h_1 = p - 2$, $h_2 = t$, and $h_3 = -t + p - 3$. Systematically exploiting transient effects in this way is an interesting direction for further work.

E. Pseudo-Affine Parameterized Looped Schedules

Here, we examine a useful generalization of APLSs that adopts pseudo-affine functions. This generalization can efficiently accommodate forms of Compaan traces that arise in practice, but do not fit well within the restrictions of the APLS formulation that we have discussed so far. In this case, the iteration count functions turn into (for a single parameter p) $\langle a_1, \ldots, a_m \rangle_p p + \langle b_1, \ldots, b_n \rangle_p$, where there are m (n) possibilities for a (b), and the choice depends on the value of p.

Example 6: Suppose that we are given a pseudo-affine loop $L = ([\langle 1, 2 \rangle_p p + \langle 2, 3 \rangle_p]A)$ such that if p is odd, then 1 will be chosen from $\langle 1, 2 \rangle_p$, and if p is even, then 2 will be chosen, and similarly, 2 for odd p and 3 for even p will be selected from $\langle 2, 3 \rangle_p$. It can be verified then that L will return $(5A)$ for $v(p) = 3$ and $(11A)$ for $v(p) = 4$.

Example 7: Suppose that the upper bound of the outer loop in Fig. 4(a) is changed to $p + 2$, rather than $2p - 4$, and assume that $v(p) \geq 8$. Then, we will obtain

$$S_{odd} = (([x, f_1]\,(2\,([f_2]B)\,([f_3]A)))\,,([f_4]B)\,,([f_5]A))$$

for odd $v(p)$, where $f_1 = f_4 = (p + 1)/2$, $f_2 = x$, $f_3 = -x + p - 3$, and $f_5 = (p - 7)/2$. For even $v(p)$, we have

$$S_{even} = (([y, g_1]\,(2\,([g_2]B)\,([g_3]A))))$$

where $g_1 = (p + 2)/2$, $g_2 = y$, and $g_3 = -y + p - 3$. The two APLSs can be consolidated into a single pseudo-affine formulation

$$S = (([t, h_1]\,(2\,([h_2]B)\,([h_3]A)))\,,([h_4]B)\,,([h_5]A))$$

where $h_1 = p/2 + \langle 1/2, 1 \rangle$, $h_2 = t$, $h_3 = -t + p - 3$, $h_4 = \langle 1/2, 0 \rangle_p p + \langle 1/2, 0 \rangle_p$, and $h_5 = \langle 1/2, 0 \rangle_p p + \langle -7/2, 0 \rangle_p$.

VII. Application: Synthesis From Kahn Process Networks

The computation model of *Kahn process networks (KPNs)* [13], [17] expresses applications in terms of distributed control and memory, and is one of various ways to model signal processing applications. The KPN model [9] assumes a network of concurrent autonomous processes that communicate in a point-to-point fashion over unbounded first-in first-out (FIFO) channels, using a blocking-read synchronization primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes. For general KPN specifications, bounded memory implementation requires some form of run-time deadlock detection, and efficient techniques have been developed for this purpose (e.g., see [24]).

To facilitate migration from an imperative application specification, which is preferred by many programmers, to a KPN specification, a set of tools, *Compaan* and *Laura* [25], is being developed, as illustrated in Fig. 5(a). This approach allows parts of an application written in a subset of MATLAB to be converted automatically to KPNs. The conversion is fast and correct-by-construction [9], [25]. The obtained KPN processes can be mapped to software or hardware.

A. Interface Control Generation

In the synthesis flow of Laura, a VHDL description of an architecture is generated from a KPN. Laura converts a process specification together with an IP core into an abstract architectural model, called a *virtual processor* [9]. Every virtual processor is composed of four units Fig. 5(b): *Execution*, *Read*, *Write*, and *Controller*. Execution units contain the computational parts of virtual processors. To communicate data on FIFO channels, *ports* are devised, which connect FIFO channels and virtual processors. Read/write units are in charge of multiplexing/demultiplexing port accesses for execution units. Controller units provide valid port access sequences, or *traces*, to facilitate computation. The determination of traces, also called *interface control generation*, in a systematic way and compact form is our focus here.

A simple approach to implementing the distributed control is to use ROM tables to store the traces. However, this strategy is impractical because of large hardware costs. To reduce the complexity, several compile time techniques are proposed to compress these tables and to keep the flexibility offered by the parametric approach [9]. In this paper, APLSs are employed to compact traces to demonstrate the effectiveness of APLS for hardware implementation.

To reduce hardware area costs of the ROM table approach, construction of looped schedules can be used. Moreover, applications specified in the KPN model may have parameters that can be configured at run time. The constructed looped schedules highly depend on the parameters values set dynamically. With the isomorphism formulation stated in Section VI for APLS, groups of isomorphic looped schedules can be summarized by single APLSs if the formulation is possible (as in Fig. 6). This

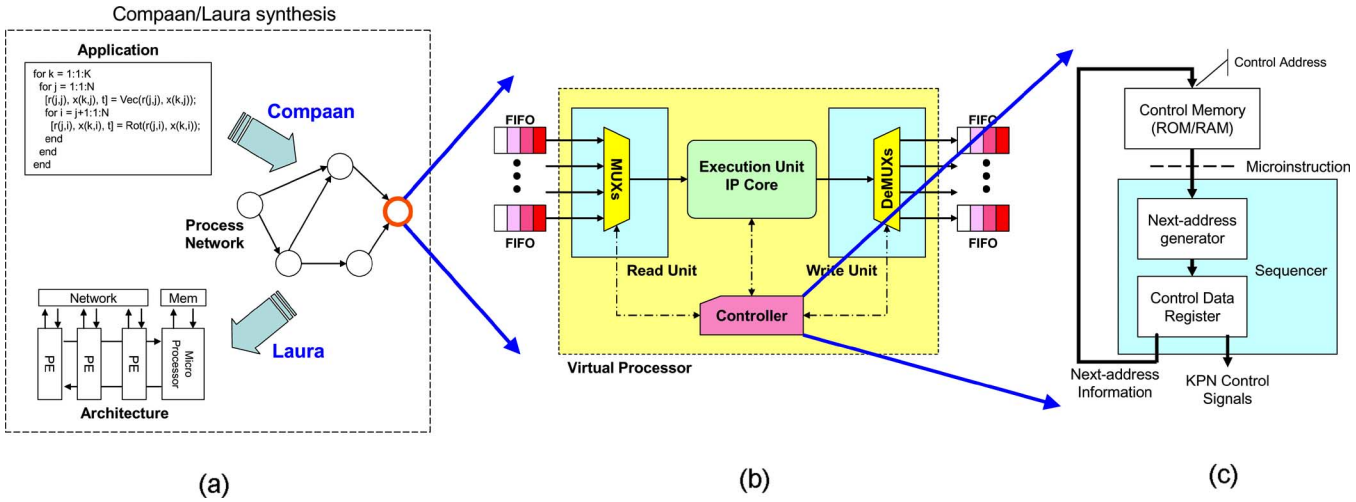


Fig. 5. Synthesis overview of APLSs for Compaan traces. (a) Compaan–Laura synthesis flow. PE stands for "processing element" and acts as a coprocessor. (b) Virtual processor implementation for a Kahn process. (c) Our APLS FPGA synthesis for controller implementation.

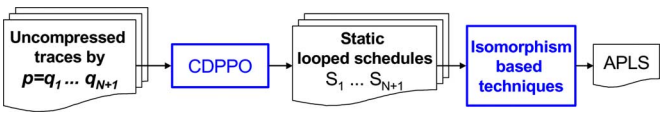


Fig. 6. APLS generation for Compaan traces.

is the way we generate the parameterizable and compact schedules, which result in significantly better performance than ROM tables.

When applied to arbitrary firing sequences (non-looped, periodic schedules) of SDF graphs, CDPPO has high computational complexity in terms of the size of the SDF graph. This is because the length of a firing sequence can grow exponentially with the size of the SDF graph. In terms of the sequence being compressed, CDPPO has polynomial complexity in terms of the length of the sequence. Thus, our proposed methods for KPN-related traces—which are applied at the level of port access traces—have polynomial complexity in the length of the given traces.

As we mention in Section VI, the APLS approach can be used on models that result in static or parameterized schedules, and this is not the case with general KPNs, where process execution may depend on I/O data. Due to restrictions that Compaan imposes on the structure of its Matlab input, the KPNs generated by Compaan belong to a restricted class of KPNs that can be statically scheduled [7]. This is why our APLS mechanism is applicable to Compaan even though it is not applicable to general KPNs. On the other hand, despite their static schedulability, KPNs generated by Compaan are more general in structure than SDF. SDF scheduling algorithms are therefore not suitable for implementing this restricted class of KPNs.

B. FPGA Setup for Controller Units

KPN control generation using APLS is implemented in a *micro-engine* architecture. Under the requirements of a virtual processor controller, the micro-engine has to perform a *for*-loop operation and generate a KPN control symbol in one cycle. As shown in Fig. 5(c), an APLS controller consists of two parts: a ROM/RAM memory and a *sequencer*. In the ROM/RAM memory is stored a compiled version of a APLS, which describes a trace using micro-instructions. The sequencer uncompresses the APLS trace and generates the desired KPN port through fetching and decoding micro-instructions from the control memory. The memory address of the next micro-instruction needs to be evaluated as well by the sequencer. To implement APLSs in FPGA hardware, the following two steps are employed.

- Symbolic program compilation: The first step involves the compilation of the input APLS using the micro-engine instruction primitives. This is done at the symbolic level.
- Hardware program generation: The second step takes the symbolic program and transforms it in a bit-stream suitable for an FPGA platform. This step takes into account the bit widths of the loop count and the symbols used in the APLS trace.

In hardware, encoding methods, such as one-hot and binary encoding, are used for the program symbols in our implementation. The choice of encoding schemes is done as a function of the dimension of the implementation and/or speed constraints.

C. Experiments

Our experiments are based on implementation costs of the controller units on an FPGA. The experiments apply the isomorphism-based APLS formulations developed in Section VI to efficiently provide for dynamic reconfiguration across scalable families of KPN implementations.

In Fig. 7, we show experimental results for FPGA synthesis on a number of applications. Here, *QR* is a matrix decomposition algorithm, and *Optical* is an image restoration algorithm [20]. For each application, particular processes that are suitable for isomorphism-based APLS formulation are selected for our experiments. We compare control memory [as shown in Fig. 5(c)] size, FPGA synthesis area, and maximum frequency for the FPGA in generating port accesses under three implementations: ROM table, RLE, and APLS. For example, virtual

Virtual Processor	Control Mem. (bytes)			FPGA Area (slices)			FPGA Freq. (MHz)		
	ROM	RLE	APLS	ROM	RLE	APLS	ROM	RLE	APLS
QR VP2	35	28	6	9	11	35	203	205	207
QR VP3	176	176	16	44	47	37	205	203	209
QR VP4	616	400	20	154	65	40	190	202	205
Optical VP3	944460	14850	160	236115	1675	132	40	110	150

Virtual Processor	FPGA Freq. (MHz)			FPGA Area (slices)		
	PPC	APLS	impr.	PPC	APLS	ovhd.
QR VP2	140	207	*47.9%*	28	35	*25.0%*
QR VP3	135	209	*54.8%*	30	37	*23.3%*
QR VP4	140	205	*46.4%*	29	40	*37.9%*
Optical VP3	129	150	*16.3%*	97	132	*36.1%*

Fig. 7. Experimental results for Compaan/KPN synthesis.

processor 3 (*VP3*) of the KPN representing Optical requires a ROM table control memory size of 944 460 bytes with parameter values set to $W = 320$ and $H = 200$. The size reduces to only 160 bytes if the APLS scheme is employed. All experiments are set up on a platform that is equipped with a *Xilinx Virtex-II 2000* device. The implementation of RLE can be viewed as *flat*, in contrast to *nested*, class 0 looped scheduling.

The obtained results are promising in terms of area and frequency. For example, the largest APLS trace occupies only 1% of the total 10752 slices of Xilinx Virtex-II 2000 FPGA while the ROM table approach, in contrast, uses approximately 9%. For the QR algorithm, we derived the ROM table for a set of typical parameters values ($N = 7$ and $K = 21$). The results with the compression technique applied show a considerable compression rate for this kind of application.

The enhancement achieved by APLS is also compared to advanced techniques experimented with in [9] (also on the Virtex-II 2000). In Fig. 7, APLS achieves up to 99% of byte savings over RLE. Regarding execution efficiency, APLS decompression generates traces with a frequency 46.4% faster over the *parameterized predicate controller (PPC)* approach, with however, an overhead of 37.9% more slices requirement.

In this section, we have shown that our proposed isomorphism-based APLS methodology is effective for interface control generation. It offers the flexibility of a parametric controller with small hardware resource requirements. However, it is possible that the APLS algorithm cannot efficiently compress some execution sequences, and this can affect controller performance. We can see this trend from Fig. 7, where the trace size difference affects the frequency of the entire design.

VIII. Application: Synthesis From Synchronous Dataflow

The SDF model [18] is an important common denominator across a wide variety of DSP design tools. An SDF program specification is a directed graph where vertices represent functional blocks (*actors*) and edges represent data dependencies. SDF actors typically correspond to DSP library modules—such as FIR and IIR filters, and FFT computations—and are activated when sufficient inputs are available. FIFO queues (or *buffers*) are usually allocated to buffer data transferred between actors. The cost of a buffer is determined by the maximum number of data items in the buffer at any time instant. In addition, for each edge e, the numbers of data values produced $prd(e)$ and consumed $cns(e)$ are fixed at compile time for each invocation of the source actor $src(e)$ and sink actor $snk(e)$, respectively. To save memory in storing actor execution sequences, previous studies have incorporated looping constructs to form

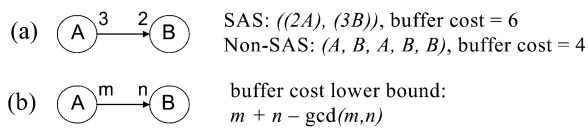


Fig. 8. Schedules and buffer costs for two-actor SDF graphs.

static looped schedules. The most compact form for such schedules, called *single appearance schedules (SAS)*, is that in which exactly one inlined version of code is allowed for each actor [2]. A two-actor SDF graph and a corresponding SAS are shown in Fig. 8(a).

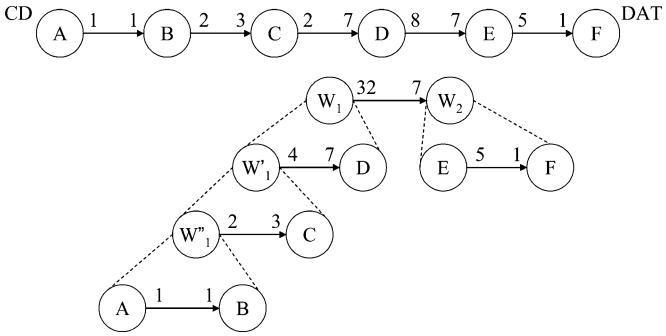
A. Minimizing Code and Data Size via PCLS

SASs in the form of static looped schedules, however, limit the potential for buffer minimization as shown in Fig. 8(a). The SAS of Fig. 8(a) has a higher buffer cost than the non-SAS does. The fixed iteration counts of static schedule loops lack the flexibility to express irregular patterns, such as the non-SAS. In contrast, the more flexible iteration control associated with the PCLS approach naturally accommodates the non-SAS in Fig. 8(a). In this case, the "instructions" in the sequences that we are trying to compress with PCLS structures correspond to the actors in the associated SDF graph.

We start by considering two-actor SDF graphs to minimize buffer costs through PCLS. A useful lower bound on the buffer memory requirement of a two-actor SDF graph, as in Fig. 8(b), is $m + n - gcd(m, n)$, and an algorithm, which we call *TASA (two-actor scheduling algorithm)*, is given in [2] to compute schedules that provably achieve this bound. Intuitively, this algorithm executes the source actor just enough times to trigger execution of the sink actor, and the sink actor executes as many times as possible (based on the available input data) before control is transferred back to the source actor. To apply this method to PCLS construction, suppose that we are given a two-actor SDF graph as shown in Fig. 8(b). Then, based on the TASA algorithm from [2], it is easily shown that depending on the values of m and n, the buffer memory lower bound $m+n-gcd(m,n)$ can be reached through the following PCLSs.

- If $m \geq n$, $PCLS = (([x_1, f_1]A([f_2]B)))$, where $f_1 = n/gcd(m,n)$ and $f_2 = \lfloor m(x_1+1)/n \rfloor - \lfloor mx_1/n \rfloor$.
- If $m \leq n$, $PCLS = (([x_1, f_1]([f_2]A)B))$, where $f_1 = m/gcd(m,n)$ and $f_2 = \lceil n(x_1+1)/m \rceil - \lceil nx_1/m \rceil$.

To extend this two-actor PCLS formulation to arbitrary acyclic graphs, we can apply the recursive graph decomposition approach in [15]. The work of [15] focuses on systematic implementation based on nested procedure calls, where both data and program memory size are considered in the optimization process. The work of [15] starts by effectively decomposing



The PCLS is

$$(([x_1,f_1]([x_2,f_2]([g_2]([x_3,f_3]([g_3]AB)C))D)([g_1](E(5F))))),\ \text{where}$$

$$f_1 = 7,\ f_2 = 4,\ f_3 = 2,$$

$$g_1 = \lfloor 32(x_1+1)/7 \rfloor - \lfloor 32x_1/7 \rfloor,$$

$$g_2 = \lceil 7(x_2+1)/4 \rceil - \lceil 7x_2/4 \rceil, \text{ and}$$

$$g_3 = \lceil 3(x_3+1)/2 \rceil - \lceil 3(x_3/2) \rceil.$$

Fig. 9. Two-actor graph decomposition for CD to DAT.

an SDF graph into a hierarchy of two-actor SDF graphs. An example of *CD (compact disc)* to *DAT (digital audio tape)* sample rate conversion is given in Fig. 9 to demonstrate this decomposition. To adapt the approach to PCLS implementation, the graph decomposition hierarchy can be mapped into a corresponding hierarchy of PCLS-based parenthesized terms.

B. Experiments

Experiments are set up to compare the results of PCLS-based inline synthesis with two other advanced techniques for joint code/data minimization, *nested procedure synthesis (NEPS)* [15] and *dynamic loop-count inline synthesis (DLC)* [23]. Our comparison is in terms of execution time and code size. Nine benchmarks available from the *Ptolemy tool* [11] are used in the experiments. The first four benchmarks are different multistage implementations of sample-rate conversion between CD and DAT formats. The other five, with labels of the form x_y_z, are for nonuniform filter banks, where the high (low) pass filters retain y/x (z/x) of the spectrum. In the PCLS-based synthesis, iteration counts are pre-computed and saved in arrays so that they can be retrieved efficiently by indexing. The target processors are from the *Texas Instruments TMS320C670x* series.

Experimental results are summarized in Fig. 10. We measure the performance ratio in terms of both execution time and code size. Formally, ratio percentages are calculated by $X/PCLS$, where X is the performance result of NEPS or DLC. A percentage larger (less) than 100% indicates that PCLS performs better (worse) than X does on that particular benchmark. PCLS synthesis demonstrates small advantages in execution time for the filter bank examples, which require longer execution latency compared to the rate conversion benchmarks. Regarding code size efficiency, PCLS demonstrates more utility (average code size reduction of 11% and 7% over NEPS and DLC, respectively). On the other hand, for arbitrary graphs that contain more than two actors, PCLS-based synthesis usually has the disadvantage of higher buffer costs compared to DLC.

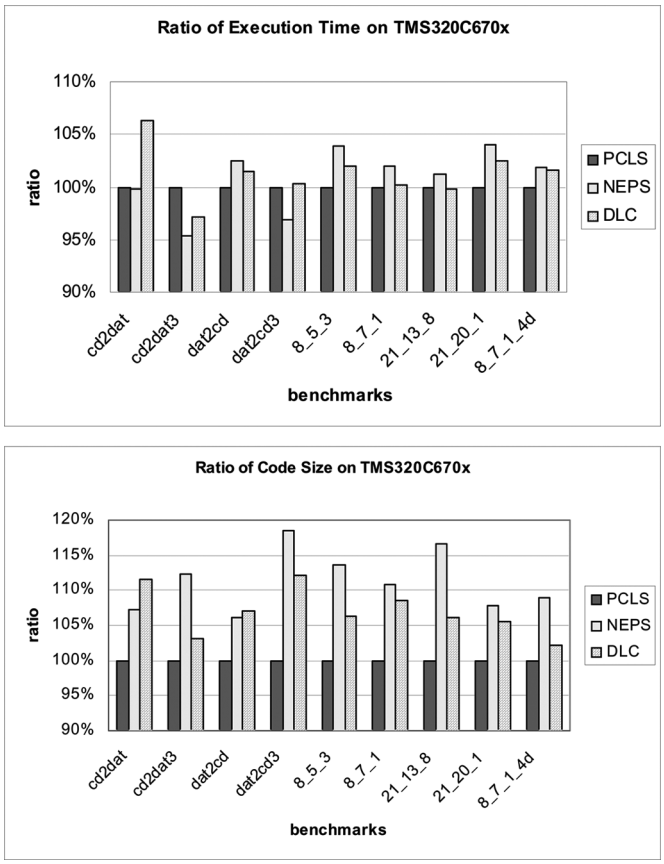


Fig. 10. Comparison of PCLS, NEPS, and DLC synthesis.

Our development of PCLS is further advantageous compared to alternative methods because it can naturally provide compaction for groups of static schedules, as demonstrated in Section VI, instead of just individual schedules in isolation. This advantage is especially useful for design space exploration, where designers may wish to experiment across a set of alternative implementations without having to resynthesize for each experiment.

IX. Summary and Future Work

This paper has focused on the motivation for formally examining broader classes of looped schedules, and on the definition and application of parameterized, constant-update looped schedules (PCLSs) for generating static execution sequences (programs). PCLSs go beyond traditional static looped schedules by making the management of loop counters more explicit. This greatly enlarges the space of execution sequences that can be compactly represented, while requiring low overhead in most implementation contexts. As the terminology in this paper suggests, there are possibilities for further enriching the classes of looped schedules under investigation (i.e., class 2, class 3, etc., looped schedules). For example, one might consider a more general class of schedules in which output values computed by "instructions" can be captured and used in the initialization or updating of loop iteration counts. Such classes of looped schedules may have the capability to express more irregular execution patterns, and more broadly, to achieve new tradeoffs among generality (the class of supported sequences), efficiency in the target

TABLE I
SUMMARY OF MAJOR NOTATIONS

Z	Set of all integers.	f	Schedule loop iteration count function.
c	Constant value.	p_i	Parameter with index i of $\text{params}(S) = \{p_1 p_2 \ldots p_n\}$.
S	Sequence or schedule.	I_i	Schedule loop iterand with index i in $L = (cI_1 I_2 \ldots I_n)$.
L	Schedule loop.	v	The assignment of values to parameters, v:params$(S) \rightarrow Z$.
P	Program.	ϕ	Isomorphic family.
α	Cost function.	θ	Isomorphic image function.

implementation, and efficiency of algorithm support in deriving the compressed representations.

Exploring the interaction of the techniques in this paper with implementation issues such as instruction-level parallelism and energy efficiency are also useful directions for further investigation.

APPENDIX

Table I summarizes major notations referenced in this paper.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. Signal Process.*, vol. 49, no. 10, pp. 2408–2421, Oct. 2001.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis From Dataflow Graphs*. Norwell, MA: Kluwer Academic, 1996.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Optimal parenthesization of lexical orderings for DSP block diagrams," in *Proc. Int. Workshop VLSI Signal Processing*, Osaka, Japan, 1995, pp. 177–186.

[4] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Tech. Rep. 124, 1994.

[5] K. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in *Proc. ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation (PLDI'99)*, Atlanta, GA, May 1999, pp. 139–149.

[6] S. Debray, W. Evans, R. Muth, and B. de Sutter, "Compiler techniques for code compression," *ACM Trans. Programm. Lang. Syst.*, pp. 378–415, 2000.

[7] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary cyclo-static dataflow counterparts," in *Proc. 17th Int. Conf. Application Specific Systems, Architectures, Processors (ASAP'06)*, Steamboat Springs, CO, Sep. 2006, pp. 186–190.

[8] E. F. Deprettere, E. Rijpkema, and P. Lieverse, "High level modeling for parallel executions of nested loop algorithms," in *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, Processors*, Boston, MA, Jul. 2000, pp. 79–91.

[9] S. Derrien, A. Turjan, C. Zissulescu, and B. Kienhuis, "Deriving efficient control in Kahn process networks," presented at the Int. Workshop Systems, Architectures, Modeling, Simulation, Samos, Greece, Jul. 2003.

[10] M. Drinic, D. Kirovski, and H. Vo, "Code optimization for code compression," in *Proc. Int. Symp. Code Generation Optimization*, San Francisco, CA, 2003, pp. 315–324.

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—The Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[12] C. W. Fraser, "Automatic inference of models for statistical code compression," in *Proc. Programming Languages Design Implementation*, 1999, pp. 242–246.

[13] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Cong. 74*, 1974, pp. 471–475.

[14] D. Kirovski, J. Kin, and W. H. Mangione-Smith, "Procedure based program compression," in *Proc. Int. Conf. Microarchitecture*, 1997, pp. 204–213.

[15] M. Ko, P. K. Murthy, and S. S. Bhattacharyya, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," in *Proc. Workshop Software Compilers for Embedded Processors*, 2004, pp. 47–61.

[16] R. Lauwereins *et al.*, "Geometric parallelism and cyclo-static data flow in GRAPE-II," in *Proc. Rapid System Prototyping*, Grenoble, France, 1994, pp. 90–107.

[17] E. A. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, pp. 773–799, 1995.

[18] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proc. IEEE*, vol. 75, pp. 1235–1245, 1987.

[19] S. Liao, "Code generation and optimization for embedded digital signal processors," Ph.D. disseratioin, MIT, Cambridge, MA, 1996.

[20] E. Memin and T. Risset, "On the study of VLSI derivation for optical flow estimation," *Int. J. Pattern Recogn. Art. Intell.*, vol. 14, no. 4, pp. 441–461, 2000.

[21] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Commun.*, vol. 38, no. 11, pp. 1917–1921, 1990.

[22] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 2, pp. 177–198, Feb. 2001.

[23] H. Oh, N. Dutt, and S. Ha, "Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs," in *Proc. Compilers, Architecture, Synthesis for Embedded Systems (CASES)*, San Francisco, CA, 2005, pp. 157–165.

[24] A. Olson and B. L. Evans, "Deadlock detection for distributed process networks," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, Mar. 2005, vol. 5, pp. 73–76.

[25] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: The Compaan/Laura approach," in *Proc. Design Automation Test in Europe (DATE)*, 2004, vol. 1, pp. 340–345.

[26] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. Int. Conf. Microarchitecture*, Portland, OR, 1992, pp. 81–91.

[27] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

Ming-Yung Ko received the B.S. degree in computer science from National Tsing-Hua University, Taiwan, R.O.C., in 1996 and the Ph.D. degree in electrical and computer engineering from the University of Maryland, College Park, in 2006.

He is currently a Senior Engineer of Sandbridge Technologies, Inc., White Plains, NY, a system design firm of software-defined radio handsets. His research interest is in the area of embedded systems design with an emphasis on software synthesis, memory management, system software, and performance profiling.

Claudiu Zissulescu received the M.Sc. degree in software engineering from Politechnica University of Bucharest.

He is now working as a Researcher with Chess BV, The Netherlands. His current research topics include hardware-software codesign, high-level synthesis techniques for parallel machines and wireless sensor networks.

Sebastian Puthenpurayil received the B.S. degree in electronics and communications engineering from the College of Engineering Trivandrum, University of Kerala, India, and the M.S. degree in electrical engineering from the University of Texas Pan American, Edinburg, TX, in 2002. He is currently working towards the Ph.D. degree in computer engineering in the Department of Electrical and Computer Engineering at the University of Maryland, College Park.

His research interests include design of energy aware algorithms, signal processing, and low-power VLSI design for embedded systems.

Shuvra S. Bhattacharyya (S'87–M'95) received the B.S. degree from the University of Wisconsin at Madison in 1987 and the Ph.D. degree from the University of California at Berkeley in 1994.

He is a Professor in the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies (UMIACS), at the University of Maryland, College Park. He is also an Affiliate Associate Professor in the Department of Computer Science. He is coauthor of two books and the author or coauthor of more than 60 refereed technical articles. His research interests include signal processing, embedded software, and hardware/software codesign. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory, San Jose, CA, and as a Compiler Developer at Kuck & Associates, Champaign, IL.

Bart Kienhuis received the M.S.E.E. and Ph.D. degrees from Delft University of Technology, Delft, The Netherlands, in 1994 and 1999, respectively.

During his Ph.D. studies, he has worked at Philips Research, Eindhoven, on a design methodology (the Y-chart approach) for high-performance video architectures for consumer products. His primary interest is in the area of embedded system design with an emphasis on hardware/software codesign, design space exploration, and performance modeling. From 1999 until 2000, he was a Postdoctoral Researcher in the group of Prof. E. A. Lee at the University of California at Berkeley. He is currently an Assistant Professor at Leiden University, Leiden, The Netherlands.

Ed F. Deprettere (F'96) was born in Roeselare, Belgium, on August 10, 1944. He received the M.Sc. degree from the University of Ghent, Ghent, Belgium, in 1968 and the Ph.D. degree from the Delft University of Technology, Delft, The Netherlands, in 1981.

From 1980 until 1999, he was a Professor at the Department of Electrical Engineering, Circuits and Systems section, Signal Processing Group. Since January 1, 2000, he has been a Professor at the Leiden Institute of Advanced Computer Sciences, Leiden University, Leiden, The Netherlands, where he is head of the Leiden Embedded Research Center. His current research interests are in system-level design of embedded systems, in particular for signal, image, and video processing applications, including wireless communications and multimedia.

Dr. Deprettere is Editor and Co-Editor of four books and several special issues of international journals. He is on the editorial board of three journals.