Parameterized Looped Schedules

Ming-Yung Ko, †Claudiu Zissulescu, Sebastian Puthenpurayil, Rami Nasr, Shuvra S. Bhattacharyya, †Bart Kienhius, †Ed Deprettere

Department of Electrical and Computer Engineering
University of Maryland at College Park, USA
{myko,purayil,rmn,ssb}@eng.umd.edu

†*Leiden Institute of Advanced Computer Science*
Leiden University, The Netherlands
{zissules,kienhous,edd}@liacs.nl

Abstract

This paper is concerned with the 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 ("instructions"). Such compaction, which can be viewed as a form of hierarchical run-length encoding, has application in many embedded software contexts, including efficient control generation for Kahn processes [6], and software synthesis for static dataflow models of computation, such as synchronous dataflow [1] and cyclo-static dataflow [3]. In this paper, we significantly generalize previous models for loop-based code compaction of DSP programs to yield a configurable code compression methodology that exhibits a broad range of achievable trade-offs. Specifically, we formally develop and apply to DSP hardware and software implementation a parameterizable loop scheduling approach with compact format, dynamic reconfigurability, and low overhead decompression.

1 Introduction

Due to tight resource constraints and the increasing complexity of applications, efficient program compression techniques are critical in the implementation of embedded 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 implementation 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 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 of our compression approach is demonstrated concretely through its application to control generation for Kahn processes [6], and to software synthesis for static dataflow models of computation [1].

2 Related Work

Sequence compression 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 run-length encoding, which replaces n repeated instances of a symbol by just a single instance of the symbol along with the repetition count n. Several bitmap file formats, e.g., *TIFF*, *BMP* and *PCX*, adopt variants of run-length encoding. More elaborate compression strategies employ "dictionary" look-up mechanisms. Here, multiple instances of a symbol sequence are replaced by smaller-sized pointers that reference a single "master copy" of the repeated sequence. The collection of master copies can therefore be viewed as a dictionary for purposes of compression. A typical example is the *LZ77* algorithm [14], variations of which are used in many data compression tools.

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.

There are several research works discussing reduction of code size through classical compiler optimizations such as strength reduction, dead code elimination, and common sub-expression elimination [5]. A particularly effective strategy is procedural abstraction [10], where procedures are created to take the place of duplicated code sequences. The work of [4] further reveals that procedural abstraction combined with classical compiler optimizations result in more

compact code size than each approach can achieve alone.

For embedded DSP design, many applications are based on dataflow models, which exhibit certain advantages compared to traditional sequential programming. Dataflow-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. The work of [1] adopts a dynamic programming approach to reformat repeated dataflow executions in a hierarchical run-length encoding style. However, the computational complexity of this strategy is relatively large. More importantly, the constraint of static and fixed iteration counts in the targeted class of looping structures significantly restricts compression results.

In this paper, we propose a flexible and parameterizable looping construct with compact format, dynamic reconfigurability, and fast decompression. The format embeds functions in describing variable lengths in a configurable run-length encoding to achieve better compression results. With reconfigurability, appropriate execution subsequences can be derived by adjusting parameter values at run time without modifying 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.

3 Background: Static Looped Schedules

We denote the set of all integers by Z, and the set of non-negative integers by Z^+. Suppose $S = (s_1, s_2, \ldots, s_n)$ 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 $(T + S)$. 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 $c(X)$ defines the cost of that instruction, whereas if X is a schedule, then $c(X)$ 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.

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 $S = (x_1, x_2, \ldots, x_n)$, where each x_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.

Given a class 0 looped schedule S, a schedule loop L is *contained* in S if for some i, x_i is a schedule loop and $x_i = L$ or L is a schedule loop that is nested within x_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 $S = (x_1, x_2, \ldots, x_n)$, we say that L_1 is *contained earlier* than L_2 in S if there exist x_i and x_j such that $i < j$, x_i contains L_1, and x_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 $((2A(3B))CD, (3A(2(3B)(2C))))$, 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 $I = (c_I X_1 X_2 \ldots X_p)$ — then $P(I)$ is defined recursively by

$$P(I) = (P(X_1) + P(X_2) + \ldots + P(X_p)) \times c .$$

Similarly, given a class 0 schedule $S = (x_1, x_2, \ldots, x_n)$, the program generated by S is (with a minor abuse of notation) denoted $P(S)$, and is given by

$$P(S) = P(x_1) + P(x_2) + \ldots + P(x_n) .$$

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 synchronous dataflow representations of DSP applications (e.g., see [1]).

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.

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 approach called *CDPPO* [2] 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.

4 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 sched-

ules 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 v. In this case, we write $f() = v$. 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 $L = ([f_L]B_L)$ or $L = ([f_L, u_L]B_L)$; 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 $\text{args}(f_L) = \{y_1, y_2, \ldots, y_m\}$.

Fact 1: The number of iterations executed by an invocation I_L of the class 1 looped schedule L is given by

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

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 $\text{body}(S) = (x_1, x_2, \ldots, x_n)$ is a finite sequence where each x_i is either an element of A or a class 1 schedule loop over A.

We say that a class 1 looped schedule S is *syntactically correct* if the following three conditions all hold.

- Every schedule loop $L = ([x_L, f_L, u_L]B_L)$ that is contained in S has a unique index x_L.
- $\{x_L \mid S \text{ contains } L\} \cap \text{params}(S) = \varnothing$; that is, the parameters of S are distinct from the loop indices.
- For each schedule 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 indices of schedule loops that lexically precede L; that is,

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

Containment of a schedule loop earlier than another schedule loop, 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.

Syntactic correctness is a necessary but not sufficient condition for validity of a class 1 looped schedule. Overall validity in general depends also on the context of the looped schedule. For example, a syntactically correct looped schedule for a synchronous dataflow graph may be invalid because the schedule is deadlocked (attempts to execute an actor before sufficient data has been produced for it).

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. 1. Using this semantics, we can define the program generated by an iterand of a class 1 schedule loop, and the pro-

gram 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, 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: Suppose $\alpha = (A, B, C, D, E, F)$, 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.

Because of their potential for parameterization, in terms of schedule parameters and loop indices, and because of their restriction that loop indices be updated by constant additions, we also refer to class 1 looped schedules as *parameterized, constant-update looped schedules* (*PCLSs*).

$x_L = x_L(0)$
$\text{limit}_L = f_L(y_1, y_2, \ldots, y_m)$
`while` $(x_L < \text{limit}_L)$
    `execute` B_L
    $x_L = x_L + u_L$
`end while`

Fig 1. A sketch of the execution of a class 1 schedule loop.

Theorem 1: Given a syntactically-correct PCLS S, and an assignment $v : \text{params}(S) \rightarrow Z$ of parameter values, the generated program $P(S, v)$ is finite.

Proof. Suppose that L is a schedule loop contained in S. Then there is a unique sequence $L_1, L_2, \ldots, L_n$, $n \geq 1$, of schedule loops contained in S such that L_1 is an iterand of S, $L_n = L$, and each L_{i+1} is an iterand of L_i. That is, $L_1, L_2, \ldots, L_{n-1}$ are the outer loops encapsulating L. Then the total number of invocations of L in an execution of S can be expressed as

$$\prod_{i=1}^{n} \textit{iterations}(L_i).$$

This follows from (1), which specifies the number of iterations executed by a given schedule loop invocation I_L. Since $u_{L_i} > 0$ and f_{L_i} is integer-valued, this number of iterations will always be finite. **QED.**

5 Affine Parameterized Looped Schedules

One useful special case of class 1 looped schedules arises when f_L is a *linear* function of $\text{args}(f_L)$. We call the special case *affine parameterized looped schedules* (*APLS*s).

The ability to parameterize iteration counts of schedule loops 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 schedule loops are equivalent in a certain structural sense. We refer to this form of equivalence between schedule loops as schedule loop *isomorphism*. Specifically, two static schedule loops L_1 and L_2 are *isomorphic* if there is a bijection f between the set of schedule loops contained in (L_1) and the set of schedule loops contained in (L_2) such that for each L in the domain of f, $L = (cI_1I_2\ldots I_m)$ and $f(L) = (dJ_1J_2\ldots J_n)$ satisfy the following three conditions: (a) L and $f(L)$ have the same number of iterands (that is, $m = n$); (b) for each i such that I_i is not a schedule loop (i.e., it is a "primitive" iterand), we have $J_i = I_i$; and (c) for each i such that I_i is a schedule loop, we have that J_i is also a schedule loop, and furthermore, I_i and J_i are isomorphic.

For each schedule loop L contained in (L_1), the mapping $f(L)$ 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 iso-

morphic if the schedule 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 schedule loops from $(1S_i)$ for some arbitrary i. Then for all $j \neq i$ and for each schedule loop L contained in $(1S_i)$, we define $f_j(L)$ to be the corresponding, structurally equivalent schedule loop in $(1S_j)$.

Using the concept of looped schedule isomorphism, we derive useful formulations in the remainder of 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 schedule 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 p_1 and p_2, and that these values of p be such that they reach beyond any transient effects (leading to negative, zero, or one-iteration loops when viewed from the final parameterized schedule). Note that functionally, a negative-iteration loop is just equivalent to a zero-iteration loop.

Let S_1 be the looped schedule instance corresponding to p_1 and let S_2 be the looped schedule instance corresponding to p_2. If S_1 and S_2 are not isomorphic, we need to increase $min(p_1, p_2)$, and try again.

Suppose now that we have an isomorphic schedule pair S_1 and S_2. We then take each loop L in S_1 and its image $f(L)$ in S_2. Let z_1 be the iteration count of L and z_2 be that of $f(L)$. We then set up the equations

$$z_1 = ap_1 + b, \text{ and } z_2 = ap_2 + b,$$

and solve these equations for a and b. We repeat this procedure for all schedule 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

$$a_1 p_1 + a_2 p_2 + \ldots + a_N p_N + b\,. \tag{2}$$

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 $q_{i,1}, q_{i,2}, \ldots, q_{i,N+1}$ be the corresponding parameter values for $p_1, p_2, \ldots, p_{N+1}$, 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 $f_i(L)$. We set up the following equations:

$$\begin{aligned}
z_1 &= a_1 q_{1,1} + a_2 q_{1,2} + \ldots + a_N q_{1,N} + b \\
z_2 &= a_1 q_{2,1} + a_2 q_{2,2} + \ldots + a_N q_{2,N} + b \\
&\ldots \\
z_{N+1} &= a_1 q_{N+1,1} + a_2 q_{N+1,2} + \ldots + a_N q_{N+1,N} + b
\end{aligned}$$

This can be expressed in matrix form as

$$\bar{z} = QA + \bar{b}\,, \tag{3}$$

where $\bar{z}$ is an $(N + 1) \times 1$ constant column vector, A is an $N \times 1$ column vector composed of the unknown a_i's, Q is an $(N + 1) \times N$ constant matrix composed of the parameter settings used in the selected schedule instances, and $\bar{b} = \lfloor b\ b\ \ldots\ b \rfloor^T$ is an $(N + 1) \times 1$ column vector obtained by replicating the unknown offset term b.

By solving (3), we obtain $a_1, a_2, \ldots, a_N$ and b to formulate the APLS parameterized schedule loop implementation of L, as represented in (2).

If a solution cannot be obtained, we can increase the selected $\{q_{i,1}, q_{i,2}, \ldots, q_{i,N+1}\}$ (to more completely bypass transient effects, as described earlier), or we may change the hypothesized number of parameters in the looped schedule.

6 Application: Kahn Process Networks

A. Background on synthesis from Kahn process networks

The computation model of the *Kahn Process Network* (*KPN*) expresses applications in terms of distributed control and memory. The KPN model of computation [6] assumes a network of con-

current autonomous processes that communicate in a point-to-point fashion over unbounded 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.

To facilitate the migration from an imperative application specification, which is preferred by many programmers, to a KPN specification, a set of tools, *Compaan* and *Laura* [12], is being developed. 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. The obtained KPN processes can subsequently be mapped either to software or hardware.

B. Control Generation

In the synthesis flow of Laura, 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* [6]. Every virtual processor is composed of four units (illustrated in Fig. 2): *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/de-multiplexing port accesses for execution units. Controller units provide valid port access sequences, or *traces*, to facilitate computation. The determination of port access sequences, also called *interface control generation*, in a systematic way and compact form is our focus in this section.

A simple approach to implementing the distributed control is to use ROM tables to store the

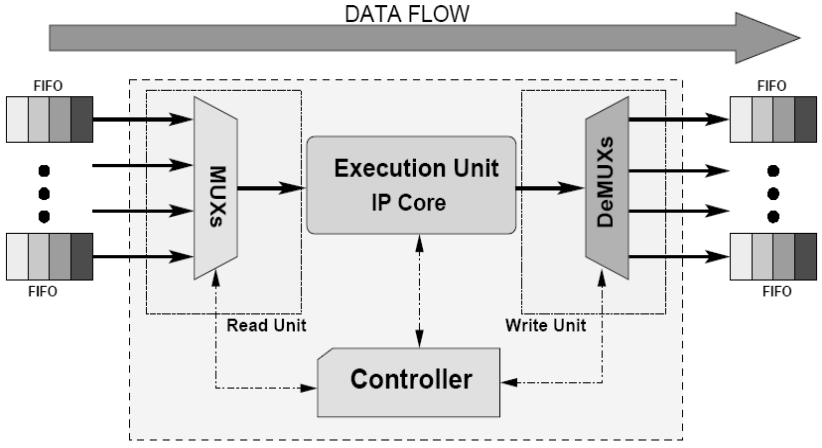

Fig 2. Hardware realization of a virtual processor.

traces. However, this strategy is impractical because of large hardware area costs. To reduce the complexity of the problem, several compile time techniques are proposed to compress these tables and to keep the flexibility offered by the parametric approach [6]. In this paper, PCLSs are employed to interface control generation to demonstrate the effectiveness of PCLS for hardware implementation.

To reduce hardware area costs of the ROM table approach, construction of class 0 looped schedules can be used. Moreover, applications specified in the KPN model may have parameters that can be configured at run time. The constructed class 0 looped schedules highly depends on the parameters values set dynamically. With the isomorphism formulation stated in Section V for APLS, groups of isomorphic class 0 looped schedules can be summarized by single APLSs if the formulation is possible. That is the way we generate the parameterizable and compact schedules, which result in significantly better performance than ROM table approach.

C. FPGA Setup for Controller Units

To reduce the memory cost, we employ a *micro-engine* architecture that computes the KPN control using PCLSs. 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. 3, a PCLS controller consists of two parts: a ROM/RAM memory and a *sequencer*. In the ROM/RAM memory is stored a compiled version of a PCLS, which describes a trace using micro-instructions. The sequencer uncompresses the PCLS trace and generates the desired KPN port through fetching and decoding micro-instructions from the control memory. The memory

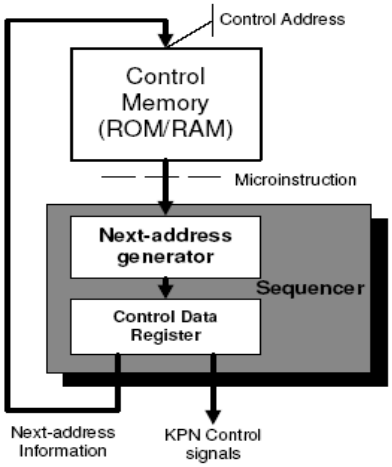

Fig 3. Microprogrammed control unit organization.

address of the next micro-instruction needs to be evaluated as well by the sequencer. To realize PCLSs in FPGA hardware implementation, the following two steps can be employed:

- Symbolic program compilation: The first step involves the compilation of the input PCLS 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 bitwidths of the loop count and the symbols used in the PCLS trace.

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

D. Experiments

Our experiments are based on implementation costs of the controller units on an FPGA. In Table I, we show the FPGA area costs (the number of FPGA slices) for a number of applications. Here, *QR* is a matrix decomposition algorithm [13], and *Optical* is a generic image restoration algorithm [11]. For each application, particular processes are selected for our experiments. We compare the size requirements under ROM table and PCLS implementations. In addition, FPGA area and maximum frequency in generating port accesses are shown. For example, virtual processor 3 (*VP3*) of the KPN representing *Optical*, requires a ROM table size of 944460 bytes with parameter values set to $W = 320$ and $H = 200$. The size reduces to only 160 bytes if the PCLS scheme is employed. All the experiments are set up on a Virtex-II 2000 equipped device.

The obtained results are promising in terms of area and frequency. For example, the largest PCLS trace occupies only 1% of the FPGA area. On the other hand, the ROM table approach uses approximately 9% of the same FPGA chip area. For the *QR* algorithm, we derived the ROM table

Table I. Experimental results on FPGA.

Virtual Processor	ROM size (bytes)	PCLS size (bytes)	PCLS freq. (MHz)	PCLS area
QR VP2	35	6	207	35
QR VP3	176	16	209	37
QR VP4	616	20	205	40
Optical VP3	944460	160	150	132

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.

In this section, we have shown that our proposed PCLS methodology is effective for interface control generation. It offers the flexibility of a parametric controller with small hardware resource requirements. However, the performance of the PCLS-based controller depends on the size of trace. It is possible that the PCLS algorithm cannot optimally compress some execution sequences, and this can in turn affect controller performance. We can see this trend from Table I where the size difference in traces affects the frequency of the entire design. For the first three traces, the frequency is approximately 205 Mhz, while for the last one the frequency drops by 26%.

7 Application: Synchronous Dataflow

A. Background on Synchronous Dataflow

The model of *synchronous dataflow* (*SDF*) [9] 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. Actors are activated when sufficient inputs are available, and FIFO queues (or *buffers*) are usually allocated to buffer data transferred between actors. 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 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 [1]. A two-actor SDF graph and a corresponding SAS are shown in Fig. 4(a).

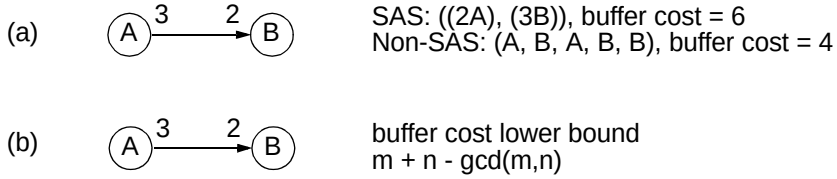

Fig 4. Examples of two-actor SDF graphs, schedules, and buffer costs.

B. Minimizing Code and Data Size via PCLS

SASs in the form of class 0 looped schedules, however, limit the potential for buffer minimization. Consider, for example, the SDF graph in Fig. 4. The SAS of Fig. 4 has a higher buffer cost than the non-SAS does. The fixed iteration counts of class 0 schedule loops lack the flexibility in expressing more irregular patterns, such as that of Fig. 4(b). In contrast, the more flexible iteration control associated with the PCLS approach naturally accommodates the non-SAS in Fig. 4(a).

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 2-actor SDF graph, as shown in Fig. 4(b), is $m + n - gcd(m, n)$, and an algorithm is given in [1] to compute schedules that 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 he source actor. This form of operation can be described more precisely in terms of the following PCLS formulation.

Theorem 2: Given a two-actor SDF graph as in Fig. 4(b), depending on the values of m and n, the buffer memory lower bound $m + n - gcd(m, n)$ can be reached through either of the following PCLSs:

- If $m \geq n$, $PCLS = (([x_1, f_1]A([x_2, f_2]B)))$, where

$$f_1 = \frac{n}{gcd(m, n)} \text{ and } f_2 = \left\lfloor \frac{(x_1 + 1)m}{n} \right\rfloor - \left\lfloor \frac{x_1 m}{n} \right\rfloor.$$

- If $m \leq n$, $PCLS = (([x_1, f_1]([x_2, f_2]A)B))$, where

$$f_1 = \frac{m}{gcd(m, n)} \text{ and } f_2 = \left\lceil \frac{(x_1 + 1)n}{m} \right\rceil - \left\lceil \frac{x_1 n}{m} \right\rceil.$$

Proof. This proof generally follows the reasoning behind the algorithm in [1] that provably reaches the minimum buffer bound for two-actor SDF graphs as in Fig. 4(b). Let us start with the case $m \geq n$. Every execution of A produces more tokens than are consumed by B. To make the smallest buffer size feasible, B must be executed repeatedly in a way that the token consumption catches up to the production as closely as possible. This behavior is carried out by the inner loop

$(A[x_2, f_2]B)$: B is consecutively executed to digest the live tokens to the full extent (i.e., any further execution of B at this point would lead to buffer underflow). The number of iterations of the outer loop is identical to the number of firings of A because A is executed only once in the inner loop. A valid SDF schedule requires that the total numbers of tokens produced and consumed have to be equal on an edge (this condition is referred to the *balance equation* for the edge). Therefore,

$$\begin{aligned} f_1 &= \frac{\text{total tokens exchanged}}{m} \\ &= \frac{mn}{gcd(m, n)} \cdot \frac{1}{m} \\ &= \frac{n}{gcd(m, n)} \end{aligned}$$

is necessary for the minimum schedule period. To determine f_2, we have to examine the total token consumption and production for an iteration. Up to the end of the $(x_1 + 1)$th iteration, there is a total of $(x_1 + 1)m$ tokens produced and $\lfloor (x_1 + 1)n/m \rfloor$ executions of B are required to maximally consume the tokens. Therefore, at the $(x_1 + 1)$th iteration, B needs $\lfloor (x_1 + 1)n/m \rfloor$ executions minus $\lfloor x_1 n/m \rfloor$ executions that have occurred in the previous iteration. Therefore, we obtain the equation of f_2, as shown above.

With a similar argument, we can derive the PCLS for the case $m \le n$. For the case of $m = n$, it does not matter which formulation ($m \ge n$ or $m \le n$) is used because both result in the same PCLS, (A, B). **QED.**

Example 4: A PCLS can be derived for the SDF graph in Fig. 4(a) by applying Theorem 2:

$$\left([x_1, 2]\left(A, \left(\left[x_2, \left\lfloor \frac{3(x_1 + 1)}{2} \right\rfloor - \left\lfloor \frac{3x_1}{2} \right\rfloor\right]B\right)\right)\right).$$

By expanding the loop hierarchy, we obtain the execution sequence (A, B, A, B, B), which results in the minimum buffer bound as shown in Fig. 4(b).

To extend this two-actor PCLS formulation to arbitrary acyclic graphs, we can apply the recursive graph decomposition approach in [8]. The work of [8] focuses on systematic implementation based on nested procedure calls, where both data and program memory space are consid-

ered in the optimization process. The work of [8] starts by decomposing an SDF graph into a hierarchy of two-actor SDF graphs. An example of CD to DAT sample rate conversion is given in Fig. 5 to demonstrate this decomposition into a hierarchy of two-actor graphs. To adapt the approach to PCLS implementation, we can map the resulting graph hierarchy into a corresponding hierarchy of PCLS-based schedule loops to derive an efficient PCLS implementation for the original SDF graph.

C. Experiments

Experiments are set up to compare the results of PCLS and nested procedure call (abbreviated as *NPC*) synthesis by means of execution time and code size. The benchmarks, *CD2DAT* and *DAT2CD*, in our experiments are obtained from the released source library of the Ptolemy project [7]. In the PCLS-based synthesis, the iteration counts in Theorem 2 are pre-computed and saved in arrays. Therefore, iteration counts are retrieved by indexing at run time. The hardware platforms in our experiments are TMS320C6xxx processor simulators from the Texas Instruments Code Composer Studio.

The experimental results are summarized in Tables II and III. We measure the percentage improvement of PCLS over NPC, $(NPC - PCLS)/NPC$, for both execution time and code size (note that buffer memory costs are identical under both approaches since the same intermediate graph hierarchy is used). A positive percentage indicates that PCLS performs better, and con-

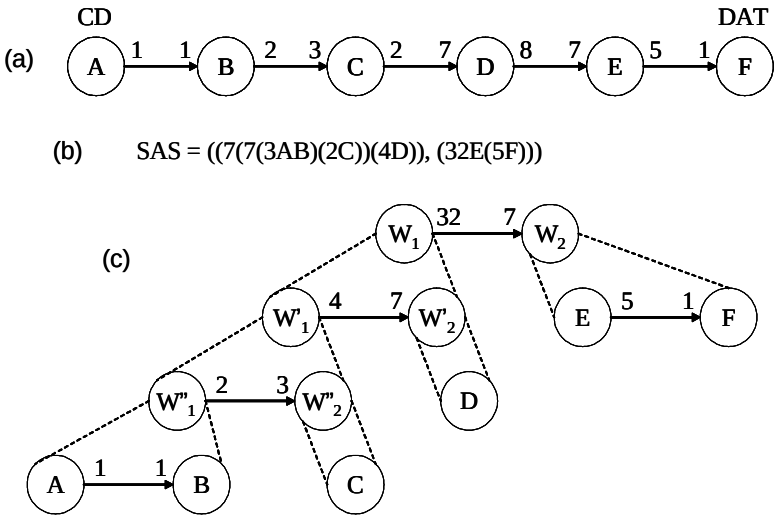

Fig 5. A CD to DAT sample rate conversion example

versely, a negative percentage indicates that NPC performs better. Moreover, C compiler optimization options are turned on selectively to stress the criticality of execution speed or code size. From both tables, PCLS always outperforms NPC in requiring less code size, especially when the "size critical" compiler configuration is selected. On the other hand, neither synthesis strategy emerges as a "clear winner" for execution time minimization. However, we also observe that the execution time overhead of PCLS-based implementation is consistently low.

8 Summary and Future Directions

This paper has focused on the motivation for 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. 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 counts, or in which the index update function can be more complex, involving possibly the indices of other loops.

Table II. Experiments on TMS320C670x processors.

	Execution Time Improvement (%)		**Code Size Improvement (%)**	
	speed critical	***size critical***	***speed critical***	***size critical***
CD2DAT	-0.86	-0.68	3.52	7.08
DAT2CD	-0.02	0.47	4.48	7.8

Table III. Experiments on TMS320C620x processors.

	Execution Time Improvement (%)		**Code Size Improvement (%)**	
	speed critical	***size critical***	***speed critical***	***size critical***
CD2DAT	0.11	4.87	2.49	5.8
DAT2CD	-0.4	-0.14	2.25	6.24

9 References

[1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Software synthesis from dataflow graphs,” Kluwer Academic Publishers, 1996.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Optimal parenthesization of lexical orderings for DSP block diagrams,” Proc. of the International Workshop on VLSI Signal Processing, Sakai, Osaka, Japan, 1995.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow,” IEEE Trans. Signal Processing, February 1996.

[4] K. D. Cooper and N. McIntosh, “Enhanced code compression for embedded RISC processors,” Proc. Conf. Programming Language Design and Implementation, May 1999.

[5] S. Debray, W. Evans, R. Muth, B. de Sutter, “Compiler techniques for code compression,” ACM Trans. Programming Languages and Systems, 2000.

[6] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis, “Deriving efficient control in Kahn process networks,” Proc. Intl. Workshop on Systems, Architectures, Modeling, and Simulation, 2003.

[7] J. Eker et al., “Taming heterogeneity — the Ptolemy approach,” Proceedings of the IEEE, Jan. 2003.

[8] M. Ko, P. K. Murthy, and S. S. Bhattacharyya, “Compact procedural implementation in DSP software synthesis through recursive graph decomposition,” Proc. Intl. Workshop on Software and Compilers for Embedded Processors, pages 47-61, September 2004.

[9] E. A. Lee and T. M. Parks, “Dataflow process networks,” Proc. of the IEEE, 83(5):773-799, May 1995.

[10] S. Liao, “Code generation and optimization for embedded digital signal processors,” PhD thesis, MIT, 1996.

[11] E. Memin and T. Risset, “On the study of VLSI derivation for optical flow estimation,” Intl. Journal of Pattern Recognition and Artificial Intelligence, 2000.

[12] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, “System design using Kahn process networks: the Compaan/Laura approach,” Proc. Design, Automation and Test in Europe Conference and Exhibition, February 2004.

[13] R. Walke, R. Smith, and G. Lightbody, “20 GFLOPS QR processor on a Xilinx Virtex-E FPGA,” Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations X, pages 300 – 310, 2000.

[14] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. Information Theory, 23:337-342, 1977.