In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, Steamboat Springs, Colorado, September 2006.

Parameterized Looped Schedules for Compact Representation of Execution Sequences

Ming-Yung Ko[1], Claudiu Zissulescu[2], Sebastian Puthenpurayil[1],
Shuvra S. Bhattacharyya[1], Bart Kienhuis[2], Ed Deprettere[2]

[1]*Dept. of Electrical and Computer Engineering, and Institute for Advanced Computer Studies*
University of Maryland at College Park, USA
{myko, purayil, ssb}@eng.umd.edu
[2]*Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands*
{zissules, kienhuis, 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. Such compaction, which can be viewed as a form of hierarchical run-length encoding (RLE), has application in many DSP system synthesis contexts, including efficient control generation for Kahn processes on FPGAs, and software synthesis for static dataflow models of computation. 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. In our experiments, this new approach demonstrates up to 99% storage saving (versus RLE) and up to 46% frequency enhancement (versus another parameterized approach) in FPGA synthesis, and an average of 11% code size reduction in software synthesis compared to existing methods for code size reduction.

1. Introduction

Due to tight resource constraints and the increasing complexity of applications, efficient program compression techniques are critical in hardware and software synthesis for embedded DSP systems. DSP subsystems 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 as well as reconfigurability during DSP system synthesis. 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 [6] on FPGAs, and to software synthesis for static dataflow models of computation, such as those developed in [3][12].

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 repeated instances of a symbol by a single instance of the symbol along with the repetition count. 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. An example is the LZ77 algorithm [15], 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 gen-

eral 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, application representations are often based on *dataflow* models of computation, 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. *Synchronous dataflow* (*SDF*) is a specialized form of dataflow that greatly facilitates static analysis for a broad class of DSP applications [9]. SDF and closely related programming models are widely used in commercial and research-oriented tools for design and implementation of DSP systems [12].

The work of [2] adopts a dynamic programming approach to reformat repeated dataflow executions in a hierarchical run-length encoding style. However, the computational complexity is relatively large, especially in system synthesis contexts. In [12], two complementary loop scheduling algorithms for dataflow-based DSP programs are proposed for joint code and data memory minimization. In the methods of [2] and [12], the constraint of static and fixed iteration counts in the targeted class of looping structures significantly restricts compression results. In [1], a meta-modeling approach is developed for incorporating dynamic reconfiguration capability into different dataflow modeling styles. When applied to SDF, this meta-modeling framework results in the *parameterized synchronous dataflow* (*PSDF*) model of computation. The developments in [1] center around a hybrid compile-time/run-time scheduling technique that is specialized to PSDF representations.

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 dataflow [3], from which static schedules can be derived. The looping construct provides compact format, dynamic reconfigurability, and fast decompression. The construct embeds functions for describing variable repetition lengths in a configurable form of run-length encoding 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.

3. Looped schedules

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, we have $p_i \in A$ for each i. 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 *loop*, L, over A to be a parenthesized term of the form $([x_L, f_L, u_L]\ B_L)$, where x_L, f_L, B_L, and u_L are, respectively, the *index*, *iteration count function*, *body*, and *index update constant* of L. The body B_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 loop. The index x_L is a symbol that represents a loop index variable associated with L in an implementation of the loop. The iteration count function is an integer-valued function $f_L(y_1, y_2, \ldots, y_m)$ defined on Z^m (Z denotes the set of all integers), where each y_i is the index of some other loop or is a configurable parameter. We represent the arguments of f_L by $\text{args}(f_L) = \{y_1, y_2, \ldots, y_m\}$. The value of f_L 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_L. It is admissible to have $m = 0$, so that f_L represents a constant value c. In this case, we write $f_L = c$.

The initial index value, denoted by $x_L(0)$, 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_L. The index update constant u_L is a positive integer that is added to x_L at the end of each iteration of L.

For brevity, we omit the initial index value from this representation. The initial index value $x_L(0)$ is specified separately when needed. Furthermore, when $u_L = 1$, we may suppress u_L from the 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 may drop the square brackets and write $L = (cB_L)$, which we denote as *static loops*.

A *looped schedule* over a finite alphabet set A is an ordered pair $S = (\text{params}(S), \text{body}(S))$. The first component $\text{params}(S) = \{p_1, p_2, \ldots, p_r\}$ is a finite set of elements called *parameters* of S, and $\text{body}(S) = (b_1, b_2, \ldots, b_n)$ is a finite sequence where each b_i is either an element of A or a loop over A. For brevity, we may write $S = (\text{body}(S))$ if $\text{params}(S) = \varnothing$. For a loop L with $B_L = I_1 I_2 \ldots I_n$, we call $S_L = (I_1, I_2, \ldots, I_n)$ the *body schedule* of L.

A special case of looped schedules is the class of *static looped schedules*. Specifically, S is a static looped schedule if $\text{params}(S)$ is empty, and every L contained in S is a static loop. Static looped schedules have been studied extensively in the context of software synthesis from synchronous dataflow representations of DSP applications (e.g., see [3][12]).

Note that in listing the set of loops that are contained in a schedule, we may need to distinguish between multiple loops that have identical iteration counts and bodies. For a looped schedule $S = (b_1, b_2, \ldots, b_n)$, a loop L is *contained* in S if for some i, $b_i = L$ or L is a loop that is nested within b_i. If L_1 and L_2 are loops that are contained in S, we say that L_1 is *contained earlier* than L_2 in S if there exist b_i and b_j such that $i < j$, b_i contains L_1, and b_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 loop L_3 such that L_1 is contained earlier than L_2 in the body schedule of L_3.

For example, consider the schedule $S = ((2A(3B))CD, (3A(2(3B)(2C))))$. Let L_1 denote the first appearance of $(3B)$, L_2 denote the second appearance of $(3B)$, L_3 denote the loop $(2A(3B))$, and L_4 denote the 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).

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.
- $\{x_L \mid S \text{ contains } L\} \cap \text{params}(S) = \varnothing$; 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 indices of loops that lexically precede L; that is,

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

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).

Intuitively, the semantics of executing a 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 Figure 1. Using this semantics, if $v : \text{params}(S) \to Z$ is an assignment of values to parameters of a looped schedule S, we write $P(S, v)$ to represent the resulting program generated by S.

As an example, suppose the set of instructions is $\{A, B, C, D, E, F\}$, and consider the 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 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

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

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 looped schedules as *parameterized, constant-update looped schedules* (*PCLSs*).

4. Affine looped schedules

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

```
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
```

Figure 1. A sketch of the execution of a loop.

The ability to parameterize iteration counts in APLSs is useful in expressing related groups of static loops. In many useful design contexts, families of static 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 loop *isomorphism*. Specifically, two static 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.

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 $(1S_j)$.

Using the concept of looped schedule isomorphism, we derive useful 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 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 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 q_1 and let S_2 be the looped schedule instance corresponding to q_2. If S_1 and S_2 are not isomorphic, we need to increase $min(q_1, q_2)$, 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 $\theta(L)$ 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 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 i th 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 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:

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

This can be expressed in matrix form as $\bar{z} = QA + \bar{b}$, 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 $\bar{b} = [b \ b \ \ldots \ b]^T$ is an $(N + 1) \times 1$ column vector obtained by replicating the unknown offset term b.

By solving the linear equations set up above, we obtain $a_1, a_2, \ldots, a_N$ and b to formulate the APLS loop implementation of L. The computational complexity in solving the linear equations is $O(N^2)$. 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.

For example, suppose we have two static loops, $L_1 = (2(2A)(1B))$ and $L_2 = (3(4A)(4B))$ induced from a single parameter p by assigning $v_1(p) = 5$ and $v_2(p) = 6$, respectively. Employing the APLS formulation techniques, the two loops can be unified to a singe expression $L = ([f_1]([f_2]A)([f_3]B))$, where $f_1(p) = p - 3$, $f_2(p) = 2p - 8$, and $f_3(p) = 3p - 14$.

5. Application: 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 [6] assumes a network of concurrent 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 migration from an imperative application specification, which is preferred by many programmers, to a KPN specification, a set of tools, *Compaan* and *Laura* [14], is being developed, as illustrated in Figure 2(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. The obtained KPN processes can be mapped to software or hardware.

5.1. 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* [6]. Every virtual processor is composed of four units (Figure 2(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/de-multiplexing port accesses for execution units. Controller units provide valid port access sequences, or

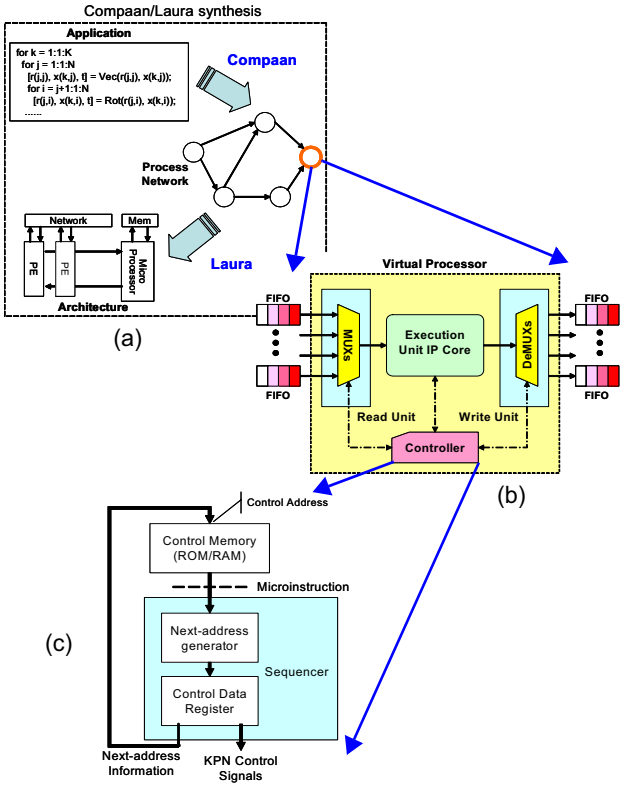


Figure 2. Overview of APLSs for Compaan traces.

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 [6]. 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 4 for APLS, groups of isomorphic looped schedules can be summarized by single APLSs if the formulation is possible (as in Figure 3). This is the way described in this section in which we generate the parameterizable and compact schedules. These schedules result in significantly better performance than ROM tables.

5.2. 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 Figure 2(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 an 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 realize APLSs in FPGA hardware implementation, the following two steps can be 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

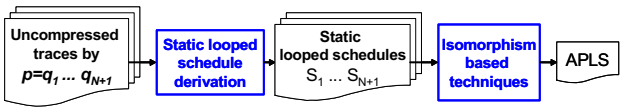


Figure 3. APLS generation for Compaan traces.

the symbolic program and transforms it to a bit-stream suitable for an FPGA platform. This step takes into account the bitwidths of the loop counts and symbols used in the APLS 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.

5.3. 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 4 to efficiently provide for dynamic reconfiguration across scalable families of KPN implementations.

In Figure 4, we show the FPGA area costs (number of FPGA slices) for a number of applications. Here, *QR* is a matrix decomposition algorithm, and *Optical* is an image restoration algorithm [11]. For each application, particular processes suitable for APLS compression techniques are selected for our experiments. We compare the area costs under ROM table and APLS implementations. Also, 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 APLS scheme is employed. All experiments are set up using a platform based on the Xilinx Virtex-II 2000 device.

The obtained results are promising in terms of area and frequency. For example, the largest APLS trace occupies only 1% of the total FPGA slices, while the ROM table approach, in contrast, uses approximately 9%. application. 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.

APLSs also achieve improvements compared to the more advanced techniques experimented with in [6] (also on the Virtex-II 2000). In Figure 4, our APLS approach achieves up to 99% of byte savings over RLE and up to 46.4% frequency improvement over the *parameterized predicate controller (PPC)* approach, with however, an overhead of 37.9% more slices required.

In this section, we have shown that our proposed 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 optimally compress some execution sequences, and this can affect controller performance. We can see this trend from Figure 4, where the trace size difference affects the frequency of the entire design. Addressing this limitation is a useful direction for further study.

	RLE (bytes)	APLS (bytes)	saving
QR VP4	400	20	95%
Optical VP3	14850	160	98.9%

	PPC (MHz)	APLS (MHz)	impr.	slices ovhd.
QR VP4	140	205	46.4%	37.9%
Optical VP3	129	150	16.3%	36.3%

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

Figure 4. Experimental results for Compaan/KPN synthesis.

6. Application: synthesis from synchronous dataflow

The *synchronous dataflow* (*SDF*) model [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 required for each actor [12]. A two-actor SDF graph and a corresponding SAS are shown in Figure 5(a).

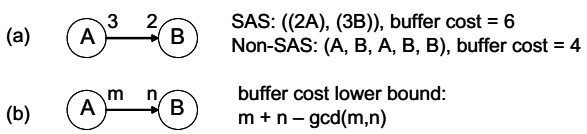


Figure 5. schedules and buffer costs for two-actor SDF graphs.

6.1. 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 Figure 5(a). The SAS of Figure 5(a) has a higher buffer cost than the non-SAS does. The fixed iteration counts of static 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 Figure 5(a).

We start by considering two-actor SDF graphs to minimize buffer costs through PCLSs. A useful lower bound on the buffer memory requirement of a two-actor SDF graph, as in Figure 5(b), is $m + n - gcd(m, n)$, and an algorithm, which we call TASA (*two-actor scheduling algorithm*), is given in [12] 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 Figure 5(b). Then, based on the TASA algorithm from [12], 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 either of 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 [8]. The work of [8] 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 [8] starts by effectively decomposing 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 Figure 6 to demonstrate this decomposition process. To adapt the approach to PCLS implementation, the graph decomposition hierarchy can be mapped into a corresponding hierarchy of PCLS-based parenthesized terms.

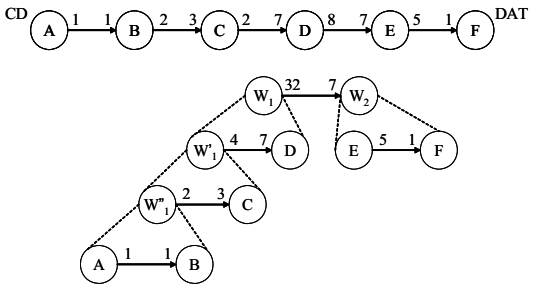
6.2. Experiments

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*) [8] and dynamic loop-count inline synthesis (*DLC*) [13]. Our comparison is in terms of execution time and code size. Nine SDF-based application models available from the Ptolemy design environment [7] are used as benchmarks in the experiments. The first four benchmarks are different multi-stage implementations of sample-rate conversion between CD and DAT formats. The other five, with labels of the form *x_y_z*, are for non-uniform 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 Figure 7. We measure the performance improvement of PCLS over NEPS and DLC for both execution time and code size. Formally, percentage numbers are calculated by $(X - PCLS)/X$, where X is the result of NEPS or DLC. A positive (negative) percentage indicates that PCLS performs better (worse). PCLS synthesis demonstrates small advantages in execution time for the filter bank examples, which require longer execution latency compared to the rate conversion benchmarks. In some experiments, PCLS does not have performance gain, due in part to the relatively small, but non-negligible overheads associated with the more complex looping structures.

On the other hand, regarding code size efficiency, PCLS demonstrates significant utility — average code size reduction of 11%, 7% over NEPS and DLC, respectively.

Our development of PCLS is further advantageous compared to alternative methods because it can naturally provide compaction for groups of static schedules, as



The PCLS is:
$(([x_1, f_1]([x_2, f_2]([g_2]([x_3, f_3]([g_3]AB)C))D)([g_1](E(5F)))))$,
where $f_1 = 7$, $f_2 = 4$, $f_3 = 2$,
$g_1 = \lfloor 32(x_1 + 1)/7 \rfloor - \lfloor 32x_1/7 \rfloor$,
$g_2 = \lceil 7(x_2 + 1)/4 \rceil - \lceil 7x_2/4 \rceil$, and
$g_3 = \lceil 3(x_3 + 1)/2 \rceil - \lceil 3(x_3/2) \rceil$.

Figure 6. Two-actor graph decomposition for CD to DAT sample rate conversion.

demonstrated in Section 4, instead of just individual schedules in isolation. This advantage is especially useful for DSP system synthesis, where designers may wish to experiment across a set of alternative implementations without having to re-synthesize for each experiment.

7. 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), which have applications in DSP system synthesis such as KPN-based FPGA synthesis and SDF-based software synthesis. 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.

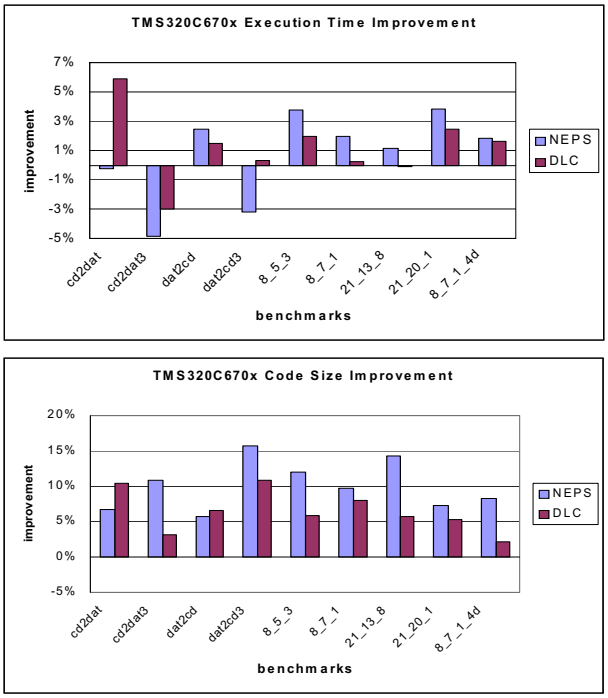


Figure 7. Comparison of PCLS, NEPS, and DLC synthesis.

8. References

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

[2] S. Bhattacharyya, P. Murthy, E. Lee, "Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams," *Proceedings of International Workshop on VLSI Signal Processing*, 1995.

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

[4] K. Cooper, N. McIntosh, "Enhanced code compression for embedded RISC processors," *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp.139-149, 1999.

[5] S. Debray, W. Evans, R. Muth, B. de Sutter, "Compiler techniques for code compression," *ACM Transactions on Programming Languages and Systems*, pp. 378-415, 2000.

[6] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis, "Deriving efficient control in Kahn process networks," *Proceedings of International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2003.

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "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," *International Workshop on Software and Compilers for Embedded Processors (SCOPES)*, pp.47-61, 2004.

[9] E. A. Lee, D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of IEEE*, 75:1235–1245, September 1987.

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

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

[12] P. K. Murthy and S. S. Bhattacharyya. *Memory Management for Synthesis of DSP Software*. CRC Press, 2006.

[13] H. Oh, N. Dutt, S. Ha, "Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs," *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 157-165, September 2005.

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

[15] J. Ziv, A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, 23(3):337-343, 1977.