Journal of VLSI Signal Processing 38, 131–146, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

The CBP Parameter: A Module Characterization Approach for DSP Software Optimization

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

PRAVEEN K. MURTHY
Fujitsu Labs of America, Sunnyvale, California, USA

Received November 11, 2002; Revised October 16, 2003; Accepted November 26, 2003

Abstract. Memory consumption is an important metric for DSP software implementation. In this paper, we develop a module characterization technique that promotes more economical use of memory resources at the system level. Our work is developed in the context of software synthesis from signal/video/image processing applications expressed as synchronous dataflow (SDF) graphs. SDF is a restricted form of dataflow where each computational module (*actor*) consumes and produces a fixed number of data values (*tokens*) on each execution. Usually, no assumption is made about when during the execution of an actor, the tokens are actually consumed and produced; the firing of an actor is treated as an atomic event for most purposes. However, we show in this paper that it is possible to concisely and precisely capture key properties pertaining to the relative times at which tokens are produced and consumed by an actor. We show this by introducing the *consumed-before-produced* (*CBP*) parameter, which provides a general method for characterizing the token transfer of an SDF actor. Good bounds on the CBP parameter can aid an SDF compiler in performing more aggressive optimizations for reducing buffer sizes on the edges between actors. We formally define the CBP parameter; derive some useful properties of this parameter; illustrate how the value of the parameter is derived by examining in detail the multirate FIR filter, which is a fundamental actor in multirate signal processing applications; and examine CBP parameterizations for several other practical SDF actors.

Keywords: array lifetime, block diagram compilers, buffer overlaying, dataflow, design methodology, embedded systems, graph coloring, lifetime analysis

1. Introduction

Block diagram environments are proving to be increasingly popular for developing DSP systems. Numerous commercial and research-oriented design tools for DSP have proliferated in recent years, such as System Canvas [1] from Angeles Design Systems, COSSAP [2] from the Aachen University of Technology (now from Synopsys), GRAPE [3] from K. U. Leuven, Ptolemy [4] from U. C. Berkeley, SPW from Cadence, and ADS from Hewlett Packard. Raising the level of abstraction in system-level design has been recognized as a key step that facilitates faster design cycles, easier retargeting, and verification. Block diagram descriptions of systems are an attractive alternative as abstract specifications to high level language (HLL) code because a block diagram system does not overspecify the system (e.g., it exposes more of the parallelism present than operational HLL code does), enables better software organization because the library blocks used are modular and reusable, and can be targeted to a variety of platforms (e.g., see [5]). However, for block diagrams to become viable as abstract specifications instead of HLLs, good synthesis flows are necessary.

In a block-diagram environment, the user connects up various blocks drawn from a library to form the system of interest. These blocks communicate with each other by writing and reading tokens (samples of data) via FIFO queues that are usually implemented as buffers. For simulation, these blocks are typically written in an HLL like C++. For software synthesis, the technique typically used is that of inline code generation: a schedule is generated, and the code generator steps through this schedule and substitutes the code for each actor that it encounters in the schedule. The code for the actor may be of two types. It may be the HLL code itself, obtained from the actor in the simulation library. The overall code may now be compiled for the appropriate target. Or the code may be hand-optimized code targeted for a particular target implementation. For programmable DSPs, this means that the actors implement their functionality through hand-optimized assembly language segments. The code-generator, after stitching together the code for the entire system then simply assembles it and the resulting machine code can be run on the DSP. This latter technique is generally more efficient for programmable DSPs because of a lack of efficient HLL DSP compilers [6].

Fortunately, block diagrams enable platform-independent coarse-grain optimizations based on knowledge of the restricted underlying models of computation; these optimizations are frequently difficult to perform for a traditional compiler. For software synthesis, block diagram synthesis flows have two major steps: scheduling and memory allocation [7]. In this paper, we develop a characterization and model of token traffic, called the CBP parameter, in the particular model of dataflow that we use, called synchronous dataflow (SDF). Characterizing this parameter exploits properties of DSP algorithms that the designer of the algorithm might be able to supply easily. Accurate knowledge of this parameter enables SDF compilers to perform much more efficient memory allocation than they have been able to do in the past. Since this characterization and these allocation techniques operate on the coarse-grain, system level description, they are somewhat orthogonal to the optimizations that might be employed by tools lower in the flow. For example, a behavioral synthesis tool has a limited view of the code, often confined to basic blocks within each block it is optimizing, and cannot make use of the global control and dataflow that our memory allocator can exploit. Similarly, a compiler for a general-purpose HLL (such as C) typically does not have the global information about application structure that our allocator has. The techniques we develop in this paper are thus complementary to the work that is being done on developing better HLL compilers for DSPs (e.g., see [8–12]). In particular, the techniques we develop operate on the graphs at a high enough level that particular architectural features of the target processor are largely irrelevant. We assume that the actor library that the code generator has access to consists of either hand-optimized assembly code, or of specifications in a high-level language like C. If the latter, then we would have to invoke a C compiler after performing the dataflow optimizations and threading the code together. Even though this might seemingly defeat the purpose of producing efficient code, since we are using a C compiler for a DSP (the compiler might not be very good as mentioned), studies have shown that for larger systems, C code produced this way compiles better than hand-written C for the entire system [13].

2. Problem Statement and Organization of the Paper

We develop a concise characterization of the precise times during which tokens are written and read by blocks in the SDF model that is used in many signal processing block diagram environments. We refer to this characterization as the CBP parameter; an accurate figure for this parameter allows an SDF memory allocator to overlay buffers more efficiently than SDF compilers have been able to in the past. Moreover, we show that if the parameter can be supplied by the designer using the analysis techniques we use in this paper, then we do not have to resort to source code analysis to deduce this parameter; source code analysis is frequently undecidable for many problems of this nature, and even if decidable, has prohibitive complexity [14]. Since we can perform this characterization in the context of a restricted subset of dataflow that is not Turing complete, our technique is feasible, practical, and efficient. SDF compilers that make use of the CBP parameter effectively are presented elsewhere [15, 16]; we do not address those techniques in this paper. Here, we focus on techniques for determining the CBP parameter.

The key limitation of the techniques in this paper is the restriction it requires on the underlying dataflow model of computation, namely the requirement that synchronous dataflow be employed. Synchronous dataflow, as discussed in Section 4, requires

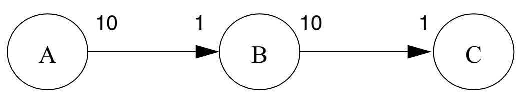


Figure 1. A simple illustration of nested iteration in a synchronous dataflow graph. The producer/consumer relationships in this example imply that actor B is executed 10 times for every execution of actor A, and actor C is executed 10 times for every execution of actor B.

that the number of data values produced and consumed by each computation be constant. This restriction precludes modeling conditionals and mutually exclusive sections of a dataflow graph (as all parts of the graph are executed unconditionally). However, nested iteration can be accommodated naturally, as illustrated in Fig. 1, and explored extensively in [17]. In the SDF graph shown in Fig. 1, each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor. Although the synchronous dataflow assumption imposes significant restrictions on program structure, the model is broad enough to cover a broad class of important signal processing applications and is the basis for several commercial design tools [9].

We review existing relevant work and contrast our work in Section 3. In Section 4 we establish notation and describe the dataflow model used for specification in DSP block diagram environments. In Section 5 we discuss the technique of buffer merging that enables more efficient buffering in SDF graphs, and we show how knowledge of the CBP parameter is necessary for the buffer merging optimization. In Section 6 we define and develop the CBP parameter. Sections 7–9 present case studies of how the CBP parameter can be computed for a number of DSP actors that are used frequently: multirate FIR filters, autocorrelators, and a "chop" actor that is used for data routing and control. In Section 10, we provide more examples in the form of a table, and finally we conclude in Sections 11 and 12.

3. Related Work

Our technique of characterizing the times at which tokens are written and read differs from the lifetime-based approach used by many HLL compilers [18, 19], SDF compilers [6], and high level synthesis tools [20, 21]. Briefly, the lifetime-based approaches try to determine the first time the token is written (**buffer start time**) and the time it is read (**buffer stop time**). A conflict graph is created based on these lifetime intervals, and graph coloring is used to obtain a memory allocation. The overlaying possibility that we expose through the CBP parameter could also be achieved using lifetime analysis techniques and graph coloring. This is the approach taken by DeGreef, Catthoor, and De Man [21] where they have developed lifetime analysis and memory allocation techniques for single-assignment, static control-flow specifications that involve explicit looping constructs, such as for loops [21], in a synthesis tool called ATOMIUM. While we cannot directly compare our approach with that of DeGreef because of differences in the starting specification used (we use a restricted subset of dataflow while they use a single-assignment static control-flow language), there are certain parallels we can draw. We have shown [7] that modeling the lifetimes of individual tokens in a multirate SDF graph can have prohibitive complexity that would preclude polynomial time solutions. Indeed, the algorithms that DeGreef et al. present have a worst case complexity that is exponential in the size of the input program [21]. Informally, the worst case exponential complexity arises in our dataflow model because the number of tokens that have to be allocated is exponential in the size of the input dataflow specification, and it is not clear whether it would be possible to somehow model all of the tokens implicitly, using a structure of size polynomial in the size of the SDF graph, while still retaining the ability to exploit the differing lifetimes of each token. Thus, the lifetime-based SDF compiler we developed [7] trades off the ability to model token lifetimes individually for a model that aggregates them and models them as blocks of tokens (with the buffer start and stop time applying to the entire block of tokens instead of any individual one) in return for efficient optimization algorithms. The drawback is that not all opportunities for overlaying memory for the tokens are made full use of. To be precise, the lifetime-based model we use [7] has to make the conservative assumption that all output buffers of an actor are simultaneously live with its input buffers. This then precludes any overlaying of output buffers with input buffers, even though subsets of those buffers might have disjoint lifetimes. However, overlaying opportunities for buffers that do not share a common actor are still present and exploited in our lifetime-based SDF compiler [7]. The algebraic approach we present in this paper allows us to refine that assumption and is able to capture the lifetimes of tokens in the buffer at the level of individual tokens so that more overlaying opportunities are exposed to the SDF compiler.

Moreover, the compiler can still make use of these enhanced overlaying opportunities through polynomial-time algorithms such as our **buffer merging** algorithms [16], and our hybrid algorithms that combine CBP-based buffer merging with lifetime-analysis techniques [15].

The CBP parameter plays a role that is somewhat similar to the array index distances derived in the in-place memory management strategies of Cathedral [22], which apply to nested loop constructs in Silage. The CBP-based buffer merging approach presented in this paper is different from the approach of Verbauwhede et al. [22] in that it is specifically targeted to the high regularity and modularity present in SDF graph implementations (at the expense of decreased generality). In particular, the overlapping of SDF input/output buffers by systematically applying CBP analysis does not emerge in any straightforward way from the more general techniques developed by Verbauwhede et al. [22]. Our form of buffer merging using the CBP parameter is especially well-suited for incorporation with the SDF vectorization techniques (for minimizing context-switch overhead) developed by Ritz et al. [23] since the absence of nested loops in the vectorized schedules allows for more flexible merging of input/output buffers. Buffer merging is also compatible with buffer access enhancements such as polyphase filter implementation [24, 25], and cyclo-static dataflow specification [26].

Vanhoof, Bolsens, and De Man have observed that in general, the full address space of an array does not always contain live data [27]. Thus, they define an "address reference window" as the maximum distance between any two live data elements throughout the lifetime of an array, and fold multiple array elements into a single window element using a modulo operation in the address calculation. The concept of the address reference window is similar to our use of the maximum number of live tokens as the size of each individual SDF buffer. The number of logically distinct memory elements (the full address space) in a buffer can be much larger than the maximum number of live tokens that reside on the buffer simultaneously [17].

Feautrier discusses the problem of counting messages in Kahn process networks [28], which is necessary for deriving producer/consumer dependencies. The problem can be solved using Ehrhardt polynomials, which are polynomials with possibly periodic coefficients. Our work on the CBP parameter represents a variation on producer/consumer dependence analysis in which the relationship being analyzed is the consumption of a value with respect to production by the same computation (actor) that performs the consumption (as opposed to the producer of the consumed value).

Darte, Schreiber, and Villard examine the problem of constructing efficient *modular allocations* to reduce memory size in application-specific processors [29]. In a modular allocation, the value computed by a statement S at the n-dimensional iteration vector I is stored in a location that can be expressed as $M_S I \bmod b_S$, where M_S is an $n \times n$ integral matrix, b_S is an n-dimensional integral vector, and the modulo operation is applied component-wise to the given vectors. In this approach, the objective is to share memory across different invocations of the same operation based on when the values produced by the operations are last used. This is in contrast to the form of sharing in CBP-based memory allocation, where overlaying is performed across two operations (or more generally, when the technique is applied multiple times in succession, to chains of operations) where data produced by one operation is overlaid with data produced by the consuming operation.

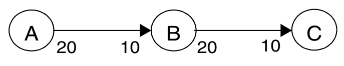
A partial summary of a preliminary version of this paper was presented in [30].

4. Notation and Background

Dataflow is a natural model of computation to use as the underlying model for a block diagram language for designing digital signal processing (DSP) systems. Functional blocks in dataflow-based, block-diagram languages correspond to vertices (**actors**) in a dataflow graph, and the connections correspond to directed edges between the actors. These edges not only represent communication channels, conceptually implemented as FIFO queues, but also establish precedence constraints. An actor **fires** in a dataflow graph by removing tokens from its input edges and producing tokens on its output edges. The stream of tokens produced this way corresponds naturally to a discrete time signal in a DSP system. In this paper, we consider a restricted form of dataflow called **synchronous dataflow** (**SDF**) [31]. In SDF, each actor produces and consumes fixed numbers of tokens, and these numbers are known at compile time. In addition, each edge has a fixed number of initial tokens, called *delays*.

Figure 2(a) shows a simple SDF graph. Given an SDF edge e, we denote the **source** actor (the actor that

(a)



(b) Valid Schedules

(1): ABCBCCC (2): A(2 B(2 C))

(3): A(2 B)(4 C) (4): A(2 BC)(2 C)

Figure 2. (a) An SDF graph. (b) Several possible schedules.

writes tokens onto an edge), **sink** actor (the actor that reads tokens from an edge), and **delay** of e by $src(e)$, $snk(e)$, and $del(e)$. Also, $prod(e)$ and $cons(e)$ denote the number of tokens **produced** onto e by $src(e)$ and **consumed** from e by $snk(e)$.

A **schedule** is a sequence of actor firings. We compile an SDF graph by first constructing a **valid schedule**—a finite schedule that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. Corresponding to each actor in the schedule, we instantiate a code block or procedure call that is obtained from a library of predefined actors. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph.

SDF graphs for which valid schedules exist are called **consistent** SDF graphs. In [31], efficient algorithms are presented to determine whether or not a given SDF graph is consistent, and to determine the minimum number of times that each actor must be fired in a valid schedule. We represent these minimum numbers of firings by a vector q_G, indexed by the actors in G (we often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the **balance equations** for G, which specify that q must satisfy

$$prod(e)q[src(e)] = cons(e)q[snk(e)], \quad \text{for every edge } e \text{ in } G. \tag{1}$$

These balance equations impose that the net data production on each dataflow edge is in balance with (equal to) the net consumption. As a result, a schedule that satisfies the balance equations can be repeated indefinitely without any unbounded accumulation of data of the graph edges.

The vector q, when it exists, is called the **repetitions vector** of G. A valid schedule then is a sequence of actor firings where each actor v is fired $q[v]$ times, and the firing sequence obeys the precedence constraints imposed by the SDF graph. For the graph in Fig. 2(a), we have $q = [1, 2, 4]$ for the actors $[A, B, C]$, and some possible schedules are shown in Fig. 2(b). The notation $(2B)$ represents the firing sequence BB. Similarly, $(2B(2C))$ represents the schedule loop with firing sequence $BCCBCC$. We define $TNSE(e)$ to be the total number of samples exchanged on edge e by actor $snk(e)$; i.e, $TNSE(e) = q[snk(e)] \cdot cons(e)$, or equivalently, from (1), $TNSE(e) = q[src(e)] \cdot prod(e)$.

As already mentioned, the first step in compiling an SDF graph is determining a schedule. Once the schedule has been determined, memory has to be allocated for the buffers in the graph. Scheduling can have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Fig. 2(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge where the size of the buffer is the maximum number of tokens queued on that edge during the schedule, are 50, 40, 60, and 50 respectively. For instance, for the first schedule, we have a maximum of 20 tokens queued on edge AB (after the firing of A), and a maximum of 30 tokens queued on edge BC after the second firing of B; hence, the total buffer memory requirement for the graph under the first schedule is 50.

Both scheduling and memory allocation present many algorithmic challenges; we tackle one particular optimization possibility for the memory allocation steps in this paper. In particular, to guide scheduling and allocation decisions, it is useful to have an accurate characterization of the interface (data production/consumption) behavior of each actor. The quantities $prod(e)$ and $cons(e)$ are examples of useful forms of interface characterization. In this paper, we develop an additional form of interface characterization, which we call the **consumed-before-produced** (**CBP**) **parameter**. The CBP parameter allows us to characterize the lifetimes of individual tokens as they are produced and consumed by SDF actors. Usually, in dataflow semantics, no assumption is made about when during the execution of an actor, the tokens are actually consumed and produced; the firing of an actor is treated as an atomic event for most purposes. However, we will show that the CBP parameter concisely and precisely captures key properties pertaining to the relative times at which tokens are produced and consumed by an actor; good bounds on the CBP parameter can aid an SDF compiler in performing more aggressive optimizations for reducing buffer sizes on the edges between actors. In this paper, we do not address the types of optimizations an SDF compiler can perform using the CBP parameter; these optimizations are presented elsewhere [15, 16]. Instead, we concentrate on techniques for computing the CBP parameter for several commonly used DSP actors. We first motivate the need for the CBP parameter by illustrating the concept of buffer merging and how buffer merging can be used to reduce buffer memory requirements.

5. Buffer Merging

Consider the second schedule in Fig. 2(b). If each buffer is implemented separately for this schedule, the required buffers on edges *AB* and *BC* will be of sizes 20 and 20, giving a total requirement of 40. Suppose, however, that it is known that *B* consumes its 10 tokens per firing *before* it writes any of the 20 tokens. Then, when *B* fires for the first time, it will read 10 tokens from the buffer on *AB*, leaving 10 tokens there. Next, it will write 20 tokens. At this point, there is a total of 30 live tokens. If we continue observing the token traffic as this schedule evolves, it will be seen that 30 is the maximum number that are live at any given time. Hence, we see that in reality, we only need a buffer of size 30 to implement *AB* and *BC*. Indeed, the diagram shown in Fig. 3 shows how the read and write pointers for actor *B* would be overlaid, with the pointers moving right as tokens are read and written. As can be seen, the write pointer, $X(w, BC)$ never overtakes the read pointer $X(r, AB)$, and the size of 30 suffices. Hence, we have merged the input buffer (of size 20) with the output buffer (of size 20) by overlapping a certain amount that is not needed because of the lifetimes of the tokens. We were able to do this because of our assumption that *B* consumes its 10 tokens per firing *before* it writes any of the 20 tokens; this is precisely the characterization we wish to develop in this paper: to determine,

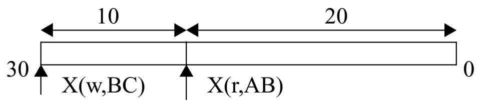


Figure 3. The merged buffer for implementing edges *AB* and *BC* in Fig. 2.

when during the firing of an actor, tokens are actually produced and consumed.

Buffer merging can be integrated synergistically with SDF lifetime analysis techniques developed in [7]. The lifetime analysis techniques developed in [7] construct a single appearance schedule optimized using a particular shared-buffer model that exploits temporal disjointedness of the buffer lifetimes. The method then constructs an intersection graph that models buffer lifetimes by nodes and edges between nodes if the lifetimes intersect in time. First-fit allocation heuristics are then used to perform memory allocation on the intersection graph. The shared buffer model used in [7] is useful for modeling the sharing opportunities that are present in the SDF graph as a whole, but is unable to model the sharing opportunities that are present at the input/output buffers of a single actor. However, the buffer merging technique enabled by CBP analysis models the input/output edge case very well, and is able to exploit the maximum amount of sharing opportunities. However, the merging process is not well suited for exploiting the overall sharing opportunities present by the graph, as that is better modeled by lifetime analysis. Hence, we have developed a bottom-up approach that combines both of these techniques, and allows maximum exploitation of sharing opportunities at both the global level of the overall graph, and the local level of an individual input/output buffer pair of an actor.

The integrated algorithm makes several passes through the graph, each time merging a suitable pair of input/output buffers. For each merge, a global memory allocation is performed using the combined lifetime of the merged buffer. If the allocation improves, then the merge is recorded. After examining each node and each pair of input/output edges, we determine whether the best recorded merge improved the allocation. If it did, then the merge is performed, and another pass is made through the graph where every node and its input/output edge pairs is examined. The algorithm stops when there is no further improvement. Further details on this algorithm are developed in [15].

6. The CBP Parameter

Informally, the CBP parameter gives the best known lower bound on the difference between the number of tokens consumed and number of tokens produced over the entire time that the actor is in the process of firing.

Formally, we say that a token is **consumed** from a memory buffer when the last access to it from the buffer is completed. For a given invocation I of an SDF actor, a given input edge α_i of the actor, and a given output edge α_o, we represent the number of tokens produced (onto α_o) and consumed (from α_i) during the time interval [0, t] by $p_I(t)$ and $c_I(t)$, respectively (time 0 corresponds to the starting time of the actor invocation, and t must be less than or equal to the completion time). If I is understood from context, we may drop the subscript I, and simply write $p(t)$ and $c(t)$.

Definition 1. Suppose that A is an actor in an SDF graph, α_i is an input edge of A, and α_o is an output edge of A. The **CBP parameter** of the pair (α_i, α_o) for the given implementation of A, denoted $\mathrm{CBP}_A(\alpha_i, \alpha_o)$, is intended to specify the best (largest) known lower bound on $c_I(t) - p_I(t), \forall I$.

Thus, if a CBP parameter has been specified by the actor programmer for (α_i, α_o), then an SDF compiler can assume that $(c_I(t) - p_I(t)) \geq \mathrm{CBP}_A(\alpha_i, \alpha_o)$ for any invocation I, and for all valid t. If no CBP parameter has been specified, a worst-case CBP parameter $\mathrm{CBP}_A(\alpha_i, \alpha_o) = -prod(\alpha_o)$ must be assumed, or the actor source code must be analyzed to try to determine a tighter bound. Such source code analysis is beyond the scope of this paper, and we simply assume the worst case bound $\mathrm{CBP}_A(\alpha_i, \alpha_o) = -prod(\alpha_o)$ when the actor programmer has not specified a CBP value.

As a simple practical example of a tight CBP bound, consider the "block addition" actor illustrated in Fig. 4(a), which inputs a block of N tokens from each input, and outputs a block of N tokens such that each ith value in the output block is the sum of the ith values in the input blocks. If the *Motorola DSP56000* code outlined in Fig. 4(b) is used to implement this actor, then it is apparent that the ith token read from each input edge is always consumed before the ith output is computed. As a result, the total number of tokens $p(t)$ produced at any given time (during the execution of a particular invocation of the actor) can never be greater than the number of tokens $c(t)$ consumed until that time from any single input edge. Thus, we are guaranteed that $c_I(t) - p_I(t) \geq 0$ and $\mathrm{CBP}_A(\alpha_i, \alpha_o) = 0$ is a valid choice.

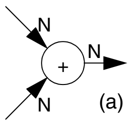
This knowledge that $c_I(t) - p_I(t) \geq 0$ allows us to fully overlay the output buffering for the code segment shown in Fig. 4(a) with either of the two input buffers. For example, if we initialize the output write pointer to the beginning of the input buffer that starts at address *inbuf1*, we are guaranteed by the relation $\mathrm{CBP}_A(\alpha_i, \alpha_o) = 0$ that the output write pointer will never "overtake" the input read pointer associated with the *inbuf1* buffer. Code for the block addition actor that incorporates this input/output overlaying is shown in Fig. 4(c). We refer to this form of buffer sharing—in which an input channel and output channel of the same actor share the same physical buffer—as **buffer merging**.

Note that we always have $\mathrm{CBP}_A(\alpha_i, \alpha_o) \leq 0$ since $c_I(0) = p_I(0) = 0$. Thus, we have the following fact.

Fact 1. *If A is an actor with input edge α_i and output edge α_o, then the value of the associated CBP parameter must satisfy*

$$(-prod(\alpha_o)) \leq \mathrm{CBP}_A(\alpha_i, \alpha_o) \leq 0. \tag{2}$$

Higher CBP values give more flexibility in buffer sharing, as will be demonstrated below, and thus, it is advantageous to specify a tight lower bound as the CBP. Due to the regularity of many DSP computations, the computation of tight CBP bounds is often



(a)

(b)

```
move #inbuf1, r0
move #inbuf2, r5
move #outbuf, r4
move x:(r0)+, x0     move y:(r5)+, a
do #N, LOOPEND
add x0, a     move x:(r0)+, x0     move y:(r5)+, a
move a, y:(r4)+
LOOPEND:
```

(c)

```
move #inbuf1, r0
move #inbuf2, r5
move r5, r4
move x:(r0)+, x0     move y:(r5)+, a
do #N, LOOPEND
add x0, a     move x:(r0)+, x0     move y:(r5)+, a
move a, y:(r4)+
LOOPEND:
```

Figure 4. A block addition actor that illustrates the derivation and exploitation of a tight CBP bound.

straightforward. In our example of Section 5, we assumed B to have $\mathrm{CBP}_B(AB, BC) = -10$; this was significantly better than the worst case assumption (when no merging would be possible) of $\mathrm{CBP}_B(AB, BC) = -20$.

Definition 2. Since $\mathrm{CBP}_A(\alpha_i, \alpha_o)$ always lies in the range

$$\{-prod(\alpha_o), -prod(\alpha_o)+1, -prod(\alpha_o)+2, \ldots, 0\},$$

the ratio of the absolute value of the CBP parameter to $prod(\alpha_o)$ is a useful gauge of the degree to which a given actor implementation A facilitates the consolidation of an input/output buffer pair. Thus, we define the **CBP efficiency** of an actor implementation with respect to the ordered pair (α_i, α_o) as the sum

$$1 + \frac{\mathrm{CBP}_A(\alpha_i, \alpha_o)}{prod(\alpha_o)}, \tag{3}$$

which is always equal to

$$1 - \frac{|\mathrm{CBP}_A(\alpha_i, \alpha_o)|}{prod(\alpha_o)} \tag{4}$$

since from Fact 1, the value of the CBP parameter is always non-positive.

Thus, the CBP efficiency is always a rational number that lies in the closed interval [0, 1]. For the example of Fig. 4, we have a CBP efficiency of unity, or 100%, since $\mathrm{CBP}_A(\alpha_i, \alpha_o) = 0$. In Section 7, we will see an example of an actor that can have an infinite range of different CBP efficiencies depending on its functional parameters.

CBP parameters can also be exploited significantly in multirate FIR filters, which are common building blocks in multirate DSP applications. As we show in the following section, a multirate FIR filter that performs a sample rate conversion of factor a/b (in reduced-fraction form) can be implemented with an efficient polyphase realization [24, 25] for which the CBP parameter can be set to

$$\mathrm{CBP}(\alpha_i, \alpha_o) = \begin{cases} 0 & \text{if } (a < b) \\ (b-a) & \text{if } (a > b) \end{cases}. \tag{5}$$

We conclude this section with a simple fact concerning CBP parameters that is useful in deriving CBP parameters for specific actor implementations.

Fact 2. *Suppose that A, α_i, and α_o are as in Definition* 1. *Given an invocation A_I of A, let t_k denote the time (relative to the beginning of A_I) at which the kth output token of A_I is produced, for $k = \{1, 2, \ldots, prod(\alpha_o)\}$. Also, define $t_0 \equiv 0$, let T_I denote the duration of A_I, and let x be a non-positive integer. Then, if $c_I(t_k) - p_I(t_k) \geq x$ for all $k \in \{0, 1, 2, \ldots, prod(\alpha_o)\}$, we are guaranteed that $c_I(t) - p_I(t) \geq x$ for all $t \in [0, T_I]$.*

The most important implication of Fact 2 is that to determine a lower bound on $c_I(t) - p_I(t)$, it suffices to examine the values of $c_I(t)$ and $p_I(t)$ only at the time instants at which output tokens are generated. In particular, we need not explicitly consider the time instants associated with consumption activity. We will exploit this simplification in Section 7.

Proof Fact 2: Since no production activity occurs between successive t_ks, we have that

$$t_k < t < t_{k+1} \Rightarrow c_I(t) - p_I(t) \geq c_I(t_k) - p_I(t_k) \quad \text{for } 0 \leq k < prod(\alpha_o). \tag{6}$$

Similarly,

$$t_{prod(\alpha_o)} < t \leq T_I \Rightarrow c_I(t) - p_I(t) \geq c_I\left(t_{prod(\alpha_o)}\right) - p_I\left(t_{prod(\alpha_o)}\right). \tag{7}$$

From (6) and (7), we can conclude that

$$\forall t^* \in [0, T_I], \exists i^* \in \{0, 1, 2, \ldots, prod(\alpha_o)\} \text{ such that } c_I(t^*) - p_I(t^*) \geq c_I(t_{i^*}) - p_I(t_{i^*}). \tag{8}$$

The desired result follows immediately from (8). □

7. Multirate FIR Filters

A multirate FIR filter actor, shown in Fig. 5(a), performs a sample rate conversion of an arbitrary rational factor u/d along with an FIR ("finite impulse response") filtering operation. Functionally, it is equivalent to the structure shown in Fig. 5(b), which contains a conventional upsampler, downsampler, and an appropriately designed single-rate FIR filter. Details on the applications and signal processing aspects of multirate FIR filters are given in [25].

The computational "core" of Fig. 5(b) is the FIR actor, which effectively forms an inner product of a vector of adjacent data samples with a vector of constant

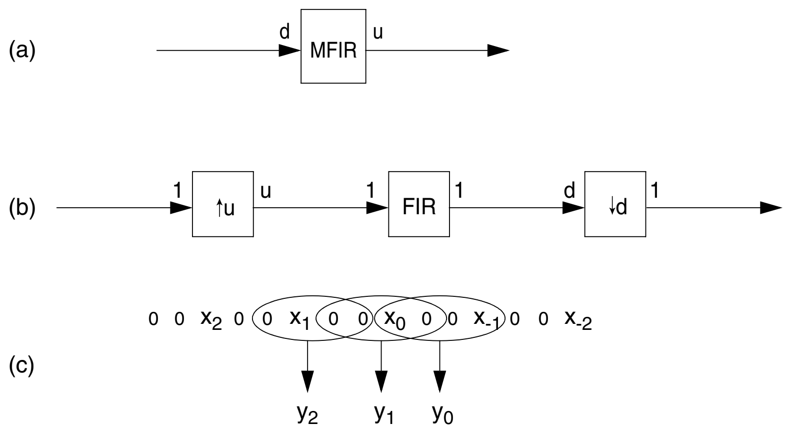


Figure 5. An example of a multirate FIR filter actor that we use to illustrate the derivation of CBP parameters.

coefficients. In this discussion, we consider the class of multirate FIR filter implementations in which the vector of "past" (previously consumed) data samples involved in the FIR inner product is maintained in a separate memory buffer that is internal to the multirate FIR actor. This is a natural approach to implementing filtering operations, and it is compatible with the concept of polyphase filter implementations in which storage and operations associated with zero-valued samples are avoided [24, 25].

In other words, we do not consider "in-place" computation of the FIR operation, where the inner product operates directly on the buffer associated with the input edge to the multirate FIR actor. This assumption is consistent with our primary objective of memory minimization since performing in-place buffering generally increases the lifetimes of the buffers on the associated edges, and thus reduces opportunities for buffer sharing [7]. The benefit of in-place buffering is that it saves the execution-time cost of having to move each data sample from the input edge buffer to the corresponding internal buffer. The problem of systematically balancing the execution-time benefits of in-place buffering for SDF graphs with the construction of memory-efficient schedules and use of buffer sharing optimizations is a useful topic for future study.

Figure 5(c) illustrates the production and consumption activity that occurs in a multirate FIR filter. In this illustration, u and d are taken to be 3 and 2, respectively, and the order of the filter is taken to be 4. The order O_M of the filter is the number of adjacent samples from the input of the FIR block of Fig. 5(b) that are involved in the computation of each output sample.

Since $u = 3$ and $d = 2$ in the illustration of Fig. 5(c), the multirate FIR filter actor consumes 2 tokens and produces 3 tokens per invocations here. The first row of symbols (zeros and x_is) shown in Fig. 5(c) represents a stream of data samples processed in a given invocation M_I of the multirate FIR filter M. The zeros shown in the stream are inserted by the logical upsampler block (labeled "↑u") in Fig. 5(b). The upsampler effectively interleaves $(u - 1)$ zeros between each pair of input tokens.

Each x_i represents the input token value at offset i relative to the beginning of M_I. Thus, x_0 and x_1 are, respectively, the first and second token values consumed by M_I; x_2 is the first token value consumed by M_{I+1}, the next invocation of M; and x_{-1} is last token value consumed by M_{I-1} (if M_I is the first invocation of M—that is, $I = 1$—then x_{-1} is part of the initial state of M). Similarly, y_0, y_1, y_2 represent the first, second and third token values produced by M_I.

The three overlapping ovals in Fig. 5(c) group sets ("windows") of $O_M = 4$ adjacent token values in the upsampled stream with the actor output tokens that are derived from them. Successive windows (windows associated with successive actor invocations) are offset by two sample positions due to the downsampler of Fig. 5(b), which has downsampling factor $d = 2$ in this example. Assuming that the output tokens y_0, y_1, y_2 are produced according to their logical ordering (as they usually are)—that is, as long as

$$(i < j) \Rightarrow (y_i \text{ is produced before } y_j) \tag{9}$$

we have the following observations from Fig. 5(c):

$$c(t_0) \geq 1, \quad c(t_1) \geq 1, \quad \text{and } c(t_2) \geq 2, \tag{10}$$

where t_0, t_1, t_2 respectively denote the times at which y_0, y_1, y_2 are produced.

From Fact 2, it follows that for the multirate FIR filter illustrated in Fig. 5(c) ($u = 3$, $d = 2$), we have that

$$c(t) - p(t) \geq min(\{(1-1), (1-2), (2-3)\}) = -1 \tag{11}$$

Thus, for this example, a CBP parameter of -1 is feasible for any implementation.

To generalize this analysis, we observe that for arbitrary u, d and O_M, the grouping of values in the upsampled stream with corresponding output tokens has the following characteristics: each pair of adjacent input tokens x_i and x_{i+1} is separated by exactly $(u - 1)$ zero-valued samples; each output token is derived from a "window" of O_M adjacent values in the upsampled stream; each successive "window" of O_M samples is shifted d units (token positions) to the left (towards increasing time) with respect to the previous window; and the first window—the window associated with output y_0—has x_0 as its left-most sample. This last characteristic depends on the *phase* setting of the multirate filter. Our analysis in this section can easily be extended to handle arbitrary, less conventional phase settings to derive CBP parameters for such cases. We omit the details in this paper.

Now let $c(t)$ and $p(t)$ denote the total number of tokens consumed and produced, respectively, during the first t time units (during the interval [0, t]) in the execution of a given invocation Z of a multirate FIR filter actor. Recall that each output token is derived from a window of O_M successive samples from the upsampled data stream illustrated in Fig. 5(b), and let L_k denote the offset, relative to x_0, of the window (i.e., the "leftmost" sample in the window) that corresponds to the kth output token. Thus, $L_1 = 0, L_2 = d, L_3 = 2d$, and so on. In other words,

$$L_k = (k-1)d \quad \text{for } k = 1, 2, \ldots, u. \tag{12}$$

Furthermore, observe that for $p(t) \geq 1$, we must have

$$L_{p(t)} \leq (c(t) - 1)u + (u - 1); \tag{13}$$

otherwise, $(c(t) + 1)$ tokens will have been consumed throughout the time interval $[0, t]$. This is because each pair of successive x_is is separated by exactly $(u - 1)$ zero-valued samples in the "internal" upsampled data stream, and we are assuming that input tokens to the multirate FIR filter are transferred to an internal buffer as soon as they are encountered (no in-place computation). In other words, the offset that corresponds to the $p(t)$th output token cannot exceed the offset associated with the first $c(t)$ tokens consumed plus the $(u - 1)$ succeeding zero-valued samples.

From (12) and (13), it follows immediately that

$$(p(t) - 1)d \leq (c(t) - 1)u + u - 1, \tag{14}$$

and although we have derived (14) under the assumption that $p(t) \geq 1$, the inequality is easily seen to hold for $p(t) = 0$ as well. This is because by definition, we have that $d \geq 1$, and $c(t) \geq 0$, and thus,

$$\begin{aligned}(c(t)-1)u + u - 1 &\geq ((c(t)-1)u + u - 1)|_{c(t)=0} \\ &= -1 \geq -d = ((p(t)-1)d)|_{p(t)=0}.\end{aligned} \tag{15}$$

We conclude that (14) holds for all t.

With some rearrangement of terms, (14) can be seen to be equivalent to

$$p(t) < 1 + \frac{c(t)u}{d}. \tag{16}$$

Now suppose, as above, that $t_0, t_1, \ldots, t_{u-1}$ denote the times at which outputs $y_0, y_1, \ldots, y_{u-1}$ respectively, are produced. Then, clearly for all i,

$$p(t_i) = (i + 1), \tag{17}$$

and combining this with (16) yields

$$i + 1 < 1 + \frac{c(t_i)}{d}u, \tag{18}$$

which is equivalent to

$$c(t_i) > \frac{di}{u}. \tag{19}$$

Thus, combining (19) and (17), we have

$$c(t_i) - p(t_i) > \frac{di}{u} - (i + 1). \tag{20}$$

From (20) and the restriction that

$$i \in \{0, 1, \ldots, (u - 1)\} \tag{21}$$

(since Z produces u output tokens), it follows that

$$c(t_i) - p(t_i) \geq 0 \quad \text{whenever } d \geq u. \tag{22}$$

On the other hand, if $u > d$, then the LHS of (20) attains its minimum value over the range (21) when $i = (u - 1)$. Thus, for $u > d$, we have

$$c(t_i) - p(t_i) > \frac{d(u-1)}{u} - u, \tag{23}$$

which is equivalent to

$$c(t_i) - p(t_i) > (d - u) - \frac{d}{u}. \tag{24}$$

Since $u > d$, and both $c(t_i)$ and $p(t_i)$ must be integers, it follows that

$$c(t_i) - p(t_i) \geq (d - u) \quad \text{whenever } u > d. \tag{25}$$

From Fact 2, we can extend the conclusions of (22) and (25) to arbitrary values of t. That is, throughout any invocation of Z, we have that

$$\begin{aligned} &(d \geq u) \Rightarrow (c(t) - p(t) \geq 0) \quad \text{and} \\ &(u > d) \Rightarrow (c(t) - p(t) \geq (d - u)). \end{aligned} \tag{26}$$

In summary, we have established the following result.

Theorem 1. *If in-place buffering is not used and output tokens are produced according to their logical ordering, then a rational, multirate FIR filter can be derived that satisfies:*

$$\text{CBP} = \begin{cases} 0 & \text{if } (u \leq d) \\ (d - u) & \text{if } (u > d) \end{cases}, \tag{27}$$

where $\frac{u}{d}$ is the reduced form of the output-to-input sample-rate conversion ratio.

From Theorem 1, we see that the CBP efficiency of a multirate FIR filter is unity (100%) if $u \leq d$; otherwise, for $u > d$, the CBP efficiency is given by

$$1 + \left(\frac{d-u}{u}\right) = \frac{u + d - u}{u} = \frac{d}{u}. \tag{28}$$

Thus, when a multirate FIR filter has an output-to-input sample-rate conversion ratio that exceeds unity, the CBP efficiency decreases monotonically with the magnitude of the conversion ratio.

8. Chop

The *chop* actor is another example of a practical actor for which CBP parameterization is useful. In this section, we consider the chop actor that is available in the Ptolemy design environment [32]. On each invocation, the chop actor reads a block of data from its input channel, and in general, produces on its output a "window" of contiguous samples from the input channel. Three parameters—the integer *offset* Δ, the boolean-valued *past-inputs* parameter, and the *production parameter* N_w—determine the size and relative position of the output window that is produced. The size of each input block is determined by the *consumption parameter* N_r. These parameters must satisfy

$$N_w + \Delta \leq N_r, \tag{29}$$

which ensures that the actor will not attempt to read samples that have not yet been produced.

If $\Delta > 0$, then the output window starts at an offset of Δ from the beginning of the input window and extends for N_w samples. The *past-inputs* parameter is not relevant in this case. If $\Delta < 0$, and *past-inputs* is false, then the first $(-\Delta)$ tokens that are produced are all zero-valued tokens, and the remaining output tokens are copies of the first $(N_w + \Delta)$ tokens in the input block. Finally, if $\Delta < 0$ and *past-inputs* is true, then the first $(-\Delta)$ tokens produced are copies of the last $(-\Delta)$ tokens from the *previous* input block. Again, the remaining output tokens are copies of the first $(N_w + \Delta)$ tokens in the input block.

Using techniques similar to those illustrated in Section 7, the chop actor can be shown to satisfy the following tight CBP parameterization:

$$\text{CBP} = \begin{cases} 0 \\ \quad \text{if } (\Delta \geq 0) \\ \Delta \quad \text{if } ((\Delta < 0) \textbf{ and } (\textit{past-input} = \text{false})) \\ (\textit{min}(\{N_r - N_w, 0\})) \\ \quad \text{if } ((\Delta < 0) \textbf{ and } (\textit{past-input} = \text{true})) \end{cases} \tag{30}$$

Dependence on the *past-inputs* parameter occurs because use of past inputs defers the removal of certain samples from the input buffer.

9. Autocorrelation

The *autocor* actor in the Ptolemy SDF DSP library [32] "estimates a certain number of samples of the autocorrelation of the input by averaging a certain number of input samples." Like the multirate FIR and chop actors, *autocor* has one input port (edge) and one output port. Two parameters control the token transfer of this actor. The first parameter N_{avg} specifies the number of input samples that are averaged, and the second parameter N_{lag} specifies the number of lags that are estimated. It is required that the value of the N_{avg} parameter be strictly greater than the value of N_{lag}. The number tokens consumed from the input edge e_i, and the number tokens produced on the output edge e_o on each invocation are given by

$$cons(e_i) = N_{\text{avg}}, \quad \text{and } prod(e_o) = 2N_{\text{lag}}. \tag{31}$$

By analyzing the definition of the Ptolemy *autocor* actor, the following tight CBP specification can be derived:

$$\text{CBP} = 1 - N_{\text{lag}}. \tag{32}$$

The associated CBP efficiency is thus given by

$$\frac{1 + N_{\text{lag}}}{2N_{\text{lag}}}. \tag{33}$$

As N_{lag} increases from its minimum possible value of 1, the CBP efficiency decreases monotonically from 100%, and asymptotically approaches 50% as $N_{\text{lag}} \to +\infty$. To get a sense of a "typical value" of CBP efficiency for this actor, observe that the default value of N_{lag} in Ptolemy is 64. From (33), this yields a CBP efficiency of 50.8%. Indeed, since usually $N_{\text{lag}} \gg 1$, the CBP efficiency of *autocor* is usually very close to 50%.

10. CBP Tables

For actors that have multiple input ports or multiple output ports, the full specification of CBP parameters takes the form of a matrix or table. Each entry of the matrix corresponds to the CBP parameter associated with the merging of a specific input port with a specific output port.

Table 1. The CBP table for the *block lattice* actor.

Input port	CBP w.r.t. output port
Coefficient input	0
Signal input	-1

Table 2. The table of CBP efficiencies for the *block lattice* actor.

Input port	CBP efficiency
Coefficient input	1.0
Signal input	$(N_B - 1)/N_B$

10.1. Block Lattice

As a simple example, consider the *block lattice* actor in Ptolemy [32], which has two input ports—the "coefficient input" and the "signal input" port—and two parameters, the block size N_B and the filter order N_o. On each invocation, N_o new filter tap values are read from the coefficient input port, a block N_B of samples is consumed on the signal input port, and a block of N_B samples is output on the output port. Tight CBP parameters for the Ptolemy implementation of *block lattice* can be specified by Table 1.

The associated CBP efficiencies can be specified in a similar manner (Table 2).

10.2. Commutator

Another example of an actor with multiple input ports is the *commutator* actor, which interleaves blocks of samples from multiple input streams onto a single output stream. This actor has three parameters—the number of input ports k, the block size N_B, and an ordering $(i_1, i_2, \ldots, i_k)$ of the input ports. On each invocation, N_B samples are consumed from each input port, and $(k \times N_B)$ samples are produced on the output port. The first N_B output samples are derived by copying N_B samples from the first input port i_1; the next block of N_B output samples is derived by copying N_B samples from the input port i_2; and so on. Since the number of samples produced on the output is significantly larger than the number consumed from any given input, the CBP efficiencies associated with this actor are relatively low. For any input port, the CBP with respect to the output port is given by

$$\text{CBP} = (1 - k)N_B, \tag{34}$$

and the CBP efficiency is given by

$$1 + \frac{(1 - k)N_B}{kN_B} = \frac{1}{k}. \tag{35}$$

10.3. Distributor

The *distributor* actor is the dual of the commutator. Like the commutator, the distributor has three parameters. These parameters specify the number of output ports (k), the block size (N_B), and an ordering $(o_1, o_2, \ldots, o_k)$ of the output ports. On a given invocation, the first (least recent) N_B samples from the input channel are copied to the first output port o_1; the next N_B input samples are copied to output port o_2; and so on. Given $i \in \{1, 2, \ldots, k\}$ and $j \in \{1, 2, \ldots, N_B\}$, the number of tokens $c(i, j)$ consumed just prior to producing the jth output sample on the ith output port is given by

$$c(i, j) = (i - 1)N_B + j. \tag{36}$$

Thus, if $c(t)$ denotes the number of tokens consumed from the input port up to time t, and $p_i(t)$ denotes the number of tokens produced on output port i up to time t, we have that

$$c(t) - p_i(t) \geq (i - 1)N_B. \tag{37}$$

From the definition of CBP, it follows that for any output port o_i,

$$\text{CBP} = 0 \tag{38}$$

is a valid CBP parameter setting for any output port with respect to the input port.

11. Summary of Derivations

To emphasize that CBP parameters may vary widely depending on the particular actor under consideration, and to juxtapose the practical examples examined in this paper, Table 3 summarizes the CBP efficiencies that we have derived. For actors that have multiple inputs or multiple outputs, we have listed the *maximum* CBP efficiency over all input/output combinations. We observe that a significant proportion of the actors examined in Table 3 admit a CBP efficiency of 100%, while the CBP efficiencies of other actors can be significantly lower and heavily parameter-dependent. To illustrate the practical impact of CBP-based analysis, we provide in Table 4 the overall buffer memory requirements from our hybrid SDF compiler that combines CBP-based buffer merging and lifetime analysis techniques for several practical systems specified as SDF graphs [15]. The systems tested here include

Table 3. A summary of the CBP parameterizations derived in this paper.

Actor	Relevant parameters	(Max.) CBP efficiency
Block addition	Block size N	1 (100%)
Multirate FIR filter	Rate conversion ratio a/b (in reduced form)	1 if $a < b$; b/a if $a > b$
Chop	Production param. N_w; consumption param. N_r; offset Δ; *past-inputs* (boolean)	1 if $\Delta \geq 0$; $1 + \Delta/N_w$ if (($\Delta < 0$) **and** (*past-inputs* = false)); $1 + \frac{\min(\{N_r - N_w, 0\})}{N_w}$ if (($\Delta < 0$) **and** (*past-inputs* = true))
Autocorrelation	Inputs to average N_{avg}; lags to estimate N_{lag}	$\frac{1+N_{\text{lag}}}{2N_{\text{lag}}}$
Block lattice	Block size N_B; filter order N_o	1
Commutator	Number of input ports k; block size N_B	$\frac{1}{k}$
Distributor	Number of output ports k; block size N_B	1

Table 4. CBP-based buffer merging and lifetime analysis applied to several practical SDF systems.

System	Non-shared	Shared	Shared and merged	Imp (%)
16qamModem	35	9	8	11.1
4pamxmitrec	49	35	18	48.6
aqmf12_3d	78	16	16	00.0
blockVox	409	135	129	04.4
cddat	264	257	205	20.2
overAddFFT	1222	514	386	24.9
phasedArray	2496	2071	1672	19.3
satrec	1542	991	773	22.0

a QMF filter bank [33], an FFT-based spectral analysis system [32], a satellite receiver [2], a CD to DAT sample-rate converter, and a phased-array system [32]. The second and third columns show the buffer memory requirements resulting from an SDF compiler that does no buffer overlaying [17], and an SDF compiler that uses lifetime analysis to do buffer overlaying [7] using the conservative assumption we described in Section 3. The fourth column shows the results of the hybrid algorithm [15]. As can be seen, the improvement over the third column is as high as 48.6% in one case with an average improvement of 18.8%. Since knowledge of the CBP parameter is a key component of the buffer merging algorithm, we conclude that knowledge and use of the CBP parameter as presented in this paper results in much more efficient SDF compilers.

12. Conclusions

The CBP parameter provides a concise and precise method for encapsulating a library developer's knowledge of DSP software functionality in a manner that is valuable for synthesis tools. Our concurrent work has demonstrated the ability to systematically exploit prespecified CBP parameters to significantly reduce memory requirements in software implementations [15, 16]. By focusing on the multirate FIR filter, we have demonstrated analysis techniques that can be used to derive tight CBP parameters from an understanding of the library function or analysis of code that implements the function. We have also given general, tight expressions for the CBP parameters of a number of additional practical DSP building blocks, which were obtained by analyzing implementations in the DSP libraries provided within the Ptolemy design environment [32]. Useful directions for further study include investigating tools to help automate the derivation of tight CBP parameters; integrating CBP-based buffering analysis, multidimensional dataflow modeling [34], and cyclo-static dataflow principles [26], which appear to have strong synergistic inter-relationships; systematically accounting for CBP parameters in the context of memory bound derivation (derivations of efficiently-computable upper bounds on memory requirements) [8]; and understanding the impact of CBP-based buffer optimization on retiming/vectorization transformations [35–37] for throughput optimization under memory capacity constraints.

Acknowledgments

This research was supported in part by the U.S. National Science Foundations (9734275) and the Advanced Sensors Collaborative Technology Alliance.

References

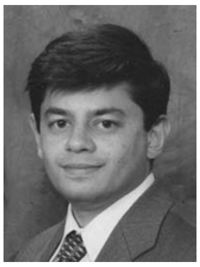
1. P.K. Murthy, E.G. Cohen, and S. Rowland, "System Canvas: A New Design Environment for Embedded DSP and Telecommunication Systems," in *Proceedings of the International Workshop on Hardware/ Software Co-Design*, April 2001.
2. S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," in *Proceedings of the International Conference on Application Specific Array Processors*, Aug. 1992, pp. 679–693.
3. R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J.V. Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, vol. 7, no. 2, 1990, pp. 32–43.
4. J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, vol. 4, 1994, pp. 155–182.
5. S. Sriram and S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc., 2000.
6. V. Zivojnovic, J.M. Velarde, C. Schlager, and H. Meyr, "DSPStone—A DSP-Oriented Benchmarking Methodology," *International Conference on Signal processing Applications and Technology*, Oct. 1994, pp. 715–720.
7. P.K. Murthy and S.S. Bhattacharyya, "Shared Buffer Implementations of Signal Processing Systems using Lifetime Analysis Techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, 2001, pp. 177–198.
8. G. Araujo and S. Malik, "Code Generation for Fixed-Point DSPs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 2, 1998, pp. 136–161.
9. S.S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software Synthesis and Code Generation for DSP, " *IEEE Transactions on*

Circuits and Systems—II: Analog and Digital Signal Processing, vol. 47, no. 9, 2000, pp. 849–875.

10. R. Leupers and P. Marwedel, "Retargetable Code Generation Based on Structural Processor Descriptions," *Journal of Design Automation for Embedded Systems*, vol. 3, no. 1, 1998, pp. 75–108.
11. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Code Optimization Techniques in Embedded DSP Microprocessors," *Journal of Design Automation for Embedded Systems*, vol. 3, no. 1, 1998, pp. 59–73.
12. P. Marwedel and G. Goossens (Eds.), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.
13. A.K. Kulkarni, A. Dube, and B.L. Evans, "Benchmarking Code Generation Methodologies for Programmable Digital Signal Processors," Technical Report. Department of Electrical and Computer Engineering, University of Texas at Austin, 1997.
14. J.E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.
15. P.K. Murthy and S.S. Bhattacharyya, "Systematic Consolidation of Input and Output Buffers in Synchronous Dataflow Specifications," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Lafayette, Louisiana, October 2000, pp. 673–682.
16. P.K. Murthy and S.S. Bhattacharyya, "Buffer Merging: A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications," in *Proceedings of the International Symposium on System Synthesis*, San Jose, California, Nov. 1999, pp. 78–84.
17. S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.
18. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1988.
19. J. Fabri, *Automatic Storage Optimization*, UMI Press, 1982.
20. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.
21. E. De Greef, F. Catthoor, and H. De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems," in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, July 1997, pp. 66–75.
22. I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man, "in-Place Memory Management of Algebraic Algorithms on Application Specific ICs," *Journal of VLSI Signal Processing*, vol. 3, no. 3, 1991, pp. 193–200.
23. S. Ritz, M. Pankert, and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," in *Proceedings of the International Conference on Application Specific Array Processors*, Oct. 1993, pp. 285–296.
24. J.T. Buck, S. Ha, D.G. Messerschmitt, and E.A. Lee, "Multirate Signal Processing in Ptolemy," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1991, pp. 1245–1248.
25. F. J. Harris, "Multirate FIR Filters for Interpolating and Desampling," in *Handbook of Digital Signal Processing*, Academic Press, 1987.
26. G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete, "Cyclo-Static Dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, 1996, pp. 397–408.
27. J. Vanhoof, I. Bolsens, and H. De Man, "Compiling Multi-Dimensional Data Streams into Distributed DSP ASIC Memory," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 1991. pp. 272–275.
28. P. Feautrier, Scheduling Kahn Process Networks. Technical report, ENS Lyon, 2002.
29. A. Darte, R. Schreiber, and G. Villard, "Lattice-Based Memory Allocation," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems*, 2003.
30. S.S. Bhattacharyya and P.K. Murthy, "The CBP Parameter—A Useful Annotation to Aid Block Diagram Compilers for DSP," in *Proceedings of the International Symposium on Circuits and Systems*, Geneva, Switzerland, 2000, pp. IV–209-IV-212.
31. E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. C–36, no. 1, 1987, pp. 24–35.
32. The Almagest, Ptolemy Reference Manual, http://ptolemy.eecs.berkeley.edu/.
33. P.P. Vaidyanathan, *Multirate Systems and Filter Banks*. Prentice Hall, 1993.
34. P.K. Murthy and E.A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, 2002, pp. 2064–2079.
35. S. Ritz, M. Willems, and H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1995, pp. 2651–2654.
36. V. Zivojnovic, S. Ritz, and H. Meyr, "Retiming of DSP Programs for Optimum Vectorization," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1994, pp. 492–496.
37. I. Hong and M. Potkonjak, "Efficient Block Scheduling to Minimize Context Switching Time for Programmable Embedded Processors," *Journal of Design Automation for Embedded Systems*, vol. 4, no. 4, 1999, pp. 311–328.
38. S.S. Bhattacharyya, J.T. Buck, S. Ha, and E.A. Lee, "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms," *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, vol. 42, no. 3, 1995, pp. 138–150.

Shuvra S. Bhattacharyya is an associate 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. Dr. Bhattacharyya is coauthor or coeditor of three books and the author or coauthor of more than 70 refereed technical articles. His research interests include VLSI signal processing, embedded software, and hardware/software co-design.

He received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley. Dr. Bhattacharyya has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and as a Compiler Developer at Kuck & Associates (Champaign, Illinois).
ssb@eng.umd.edu



Praveen K. Murthy is a member of research staff in the advanced CAD research group at Fujitsu Labs of America (FLA). His research interests span all areas of high level system design, including formal verification, validation, synthesis, simulation, techniques for optimized software and hardware code generation, semantics of different models of computation, and system level design software environments. He is a senior member of the IEEE.

He received his B.S.E.E degree from the Georgia Institute of Technology in 1989, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1993 and 1996 respectively. Prior to joining FLA, he has been with Angeles Design Systems, and Cadence Design Systems. He has also consulted for Berkeley Design Technologies Inc. in the area of DSP architectures and tools. He has co-authored numerous referred papers, including the book "Software synthesis from dataflow graphs" by Kluwer Academic Publishers.
pmurthy@fla.fujitsu.com