The CBP Parameter — a Useful Annotation to Aid SDF Compilers[1]

Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park
ssb@eng.umd.edu

Praveen K. Murthy
Angeles Design Systems
San Jose, California
pmurthy@angeles.com

Abstract

Memory consumption is an important metric for DSP software implementation [11]. In this paper, we develop a module characterization technique that promotes more economical use of memory resources at the system level. Our work is 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 firing. 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 report 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.

1. Technical report UMIACS-TR-99-56, Institute for Advanced Computer Studies, University of Maryland, College Park, 20742, September, 1999. S. S. Bhattacharyya was supported in this work by the US National Science Foundation (CAREER, MIP9734275) and Northrop Grumman Corp.

1 Introduction

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) [8]. 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*. SDF is used in numerous commercial and research-oriented design tools for DSP, such as COSSAP [12] from the Aachen University of Technology (now from Synopsys), GRAPE [7] from K. U. Leuven, Ptolemy [5] from U. C. Berkeley, DSP Canvas from Angeles Design Systems, SPW from Cadence, and ADS from Hewlett Packard.

2 Notation and background

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor. Given an SDF edge e, we denote the source actor, sink actor, 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 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 [8], 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

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

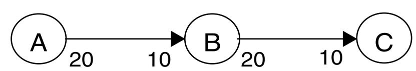
The vector q, when it exists, is called the **repetitions vector** of G.

3 Constructing memory-efficient loop structures

In [1, 2], the concept and motivation behind **single appearance schedules** (**SAS**) has been defined and shown to yield an optimally compact inline implementation of an SDF graph with regard to code size (neglecting the code size overhead associated with the loop control). An SAS is a *looped schedule* in which each actor appears only once. A looped schedule is a schedule that employs a parenthesized *schedule loop* notation to organize repetitive execution sequences into looping constructs. Figure 1 shows an SDF graph, and valid looped schedules for it. Here, the schedule loop $(2B)$ represents the firing sequence BB. Similarly, $(2B(2C))$ is a schedule loop with the firing sequence $BCCBCC$. Schedules 2 and 3 in figure 1 are single appearance schedules since actors A, B, C appear only once. An SAS like the third one in Figure 1(b) is called **flat** since it does not have any nested loops. In general, there can be exponentially many ways of nesting loops in a flat SAS [2].

Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Figure 1(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 40, 60, and 50 respectively.

(a)



Valid Schedules

(b)

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

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

Fig 1. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

To guide scheduling 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**.

4 The CBP parameter

We say that a token is *consumed* from a memory buffer when the last access to it from the buffer is completed. Also, 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 $c_I(t) - p_I(t)$.

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. 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.

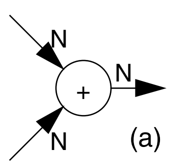
Fact 1: If A is an actor with input edge α_i and output edge α_o, then the value of the associated CBP parameter must satisfy

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

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 straightforward.

As a simple example of a tight CBP bound, consider the "block addition" actor illustrated in Figure 2(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 Figure 2(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 produced $p(t)$ 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 Figure 2(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 Figure 2(c). We refer to this form of



(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:
```

Fig 2. A block addition actor that illustrates the derivation and exploitation of a tight CBP bound.

buffer sharing — in which an input channel and output channel of the same actor share the same physical buffer — as **buffer merging**.

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 $prod(\alpha_0)$ is a useful gauge of the degree to which a given actor implementation facilitates the consolidation of an input/output buffer pair. Thus, we define the **CBP efficiency** of an actor implementation A with respect to the ordered pair (α_i, α_o) as the sum

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

which is always equal to

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

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

Thus, the CBP efficiency is always a non-negative rational number that lies in the closed interval $[0, 1]$. For the example of Figure 2, we have a CBP efficiency of unity, or 100%, since $\mathrm{CBP}_A(\alpha_i, \alpha_o) = 0$. In Section 5, 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 $\frac{a}{b}$ (in reduced-fraction form) can be implemented with an efficient polyphase realization [2, 3] for which the CBP parameter can be set to

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

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_i denote the time (relative to the beginning of A_I) at which the ith output token of A_I is produced, for $i = \{1, 2, \ldots, prod(\alpha_0)\}$. 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 $c_I(t_i) - p_I(t_i) \geq x$ for all $i \in \{0, 1, 2, \ldots, prod(\alpha_0)\}$, 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 5.

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

$$t_i < t < t_{i+1} \Rightarrow c_I(t) - p_I(t) \geq c_I(t_i) - p_I(t_i) \text{ for } 0 \leq i < prod(\alpha_0). \qquad \textbf{(EQ 6)}$$

Similarly,

$$t_{prod(\alpha_0)} < t \leq T_I \Rightarrow c_I(t) - p_I(t) \geq c_I(t_{prod(\alpha_0)}) - p_I(t_{prod(\alpha_0)}). \qquad \textbf{(EQ 7)}$$

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

$$\forall t^* \in [0, T_I], \exists i^* \in \{0, 1, 2, \ldots, prod(\alpha_0)\} \text{ such that } c_I(t^*) - p_I(t^*) \geq c_I(t_{i^*}) - p_I(t_{i^*}). \qquad \textbf{(EQ 8)}$$

The desired result follows immediately from (8). **QED**.

5 Multirate FIR filters

A multirate FIR filter actor, shown in Figure 3(a), performs a sample rate conversion of an arbitrary rational factor $\frac{u}{d}$ along with an FIR ("finite impulse response") filtering operation. Functionally, it is equivalent to the structure shown in Figure 3(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 [6].

The computational "core" of Figure 3(b) is the FIR actor, which effectively forms an inner product of a vector of adjacent data samples with a vector of constant 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 [6][4].

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 [10]. 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 compact looped schedules and buffer sharing is a useful topic for future study.

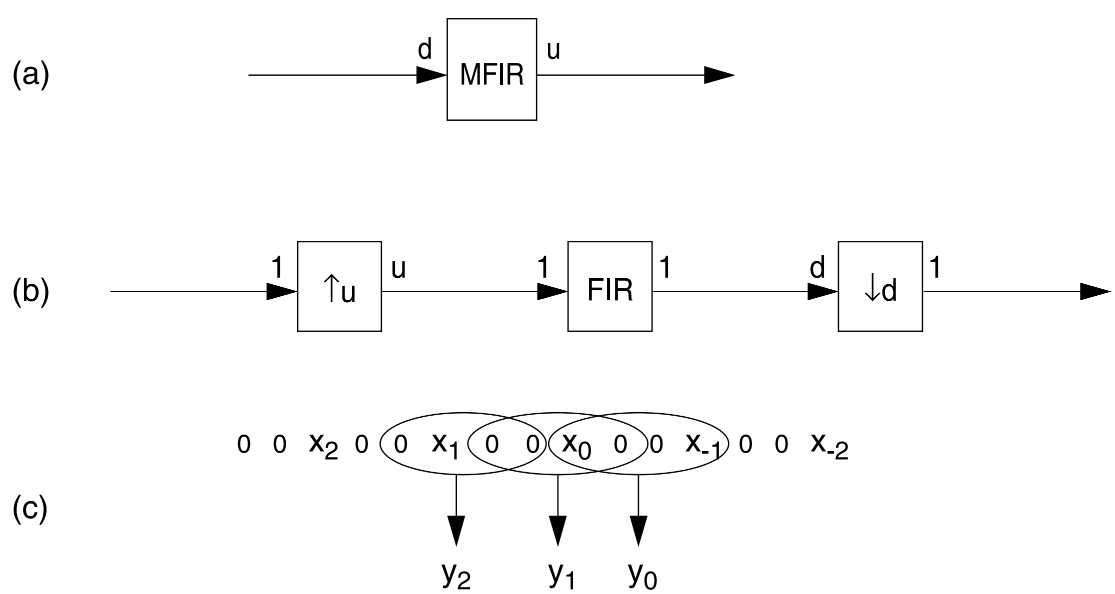


Fig 3. An example of a multirate FIR filter actor that we use to illustrate the derivation of CBP parameters.

Figure 3(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 Figure 3(b) that are involved in the computation of each output sample.

Since $u = 3$ and $d = 2$ in the illustration of Figure 3(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 Figure 3(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 Figure 3(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 Figure 3(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 Figure 3(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

$$(i < j) \Rightarrow (y_i \text{ is produced before } y_j) \text{ —} \qquad \textbf{(EQ 9)}$$

we have the following observations from Figure 3(c):

$$c(t_0) \geq 1 \text{, } c(t_1) \geq 1 \text{, and } c(t_2) \geq 2 \text{,} \qquad \textbf{(EQ 10)}$$

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

It follows that for the multirate FIR filter illustrated in Figure 3(c) ($u = 3$, $d = 2$), we have that

$$c(t) - p(t) \geq max(\{(1-1), (1-2), (2-3)\}) = 0. \qquad \textbf{(EQ 11)}$$

Thus, for this example, a CBP parameter of 0 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[1].

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 M^* 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 Figure 3(b), and let L_k denote the offset, relative to x_0, of 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,

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

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

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

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).

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

---

1. 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.

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

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

$$(c(t) - 1)u + u - 1 \ge ((c(t) - 1)u + u - 1)\big|_{c(t) = 0} = -1 \ge -d = ((p(t) - 1)d)\big|_{p(t) = 0}. \tag{EQ 15}$$

We conclude that (14) holds for all t.

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

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

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,

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

and combining this with (16) yields

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

which is equivalent to

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

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

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

From (20) and the restriction that

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

(since M^* produces u output tokens), it follows that

$$c(t_i) - p(t_i) \geq 0 \text{ whenever } d \geq u. \qquad \textbf{(EQ 22)}$$

On the other hand, if $u > d$, then the LHS of (19) attains its minimum value over the range (21) when $i = (u-1)$. Thus, for $u > d$, we have

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

which is equivalent to

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

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

$$c(t_i) - p(t_i) \geq (d-u) \text{ whenever } u > d. \qquad \textbf{(EQ 25)}$$

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

$$(d \geq u) \Rightarrow (c(t) - p(t) \geq 0) \text{ and } (u > d) \Rightarrow (c(t) - p(t) \geq (d-u)). \qquad \textbf{(EQ 26)}$$

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 from the following relations:

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

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

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

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.

6 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 [5]. 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

$$N_w + \Delta \le N_R, \qquad \textbf{(EQ 29)}$$

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 5, the implementation of the chop actor in Ptolemy can be shown to satisfy the following tight CBP parameterization:

$$\mathrm{CBP} = \begin{cases} 0 \text{ if } (\Delta \ge 0) \\ \Delta \text{ if } ((\Delta < 0) \textbf{ and } (\textit{past-inputs} = \text{false})) \\ (\mathit{min}(\{N_r - N_w, 0\})) \text{ if } ((\Delta < 0) \textbf{ and } (\textit{past-inputs} = \text{true})) \end{cases} . \qquad \textbf{(EQ 30)}$$

7 Autocorrelation

The *autocor* actor in the Ptolemy SDF DSP library "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

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

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

$$\mathrm{CBP} = 1 - N_{\mathrm{lag}}. \quad \textbf{(EQ 32)}$$

The associated CBP efficiency is thus given by

$$\frac{1 + N_{\mathrm{lag}}}{2N_{\mathrm{lag}}}. \quad \textbf{(EQ 33)}$$

As N_{lag} increases from its minimum possible value of 1, the CBP efficiency decreases monotonically from 100%, and asymptotically approaches 50% as $N_{\mathrm{lag}} \to +\infty$. 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 $N_{\mathrm{lag}} \gg 1$, the CBP efficiency of *autocor* is usually very close to 50%.

8 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.

8.1 Block lattice

As a simple example, consider the *block lattice* actor in Ptolemy, 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 of N_B 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 the following table:

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

Input port	CBP w.r.t. output port
coefficient input	0
signal input	-1

The associated CBP efficiencies can be specified in a similar manner:

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

Input port	CBP efficiency
coefficient input	1.0
signal input	$(N_B - 1)/N_B$

8.2 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

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

and the CBP efficiency is given by

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

8.3 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

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

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

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

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

$$\mathrm{CBP} = 0 \tag{EQ 38}$$

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

9 Summary of derivations

To emphasize that CBP parameters may vary widely depending on the particular actor

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** $(\textit{past-inputs} = \text{false}))$; $1 + \frac{min(\{N_r - N_w, 0\})}{N_w}$ if $((\Delta < 0)$ **and** $(\textit{past-inputs} = \text{true}))$.
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

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.

10 Related Work

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 [16], which apply to nested loop constructs in Silage. The CBP-based buffer merging approach presented in this paper is different from the approach of [16] 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 in [16]. Our form of buffer merging is especially well-suited for incorporation with the SDF vectorization techniques (for minimizing context-switch overhead) developed at the Aachen University of Technology [13] 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 [4, 6], and cyclo-static dataflow specification [3].

Vanhoof, Bolsens and H. De Man have observed that in general, the full address space of an array does not always contain live data [15]. 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 for an edge e is equal to $q(snk(e))cons(e)$, which can be much larger than the maximum number of live tokens that reside on e simultaneously [2].

11 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 previous work has demonstrated the ability to systematically exploit pre-specified CBP parameters to significantly reduce memory requirements in software implementations [9]. In this paper we have discussed the derivation of CBP parameters for individual DSP functions. 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 for a number of additional practical DSP building blocks, which were obtained by analyzing implementations in the DSP libraries provided within the Ptolemy design environment [5]. A useful direction for further study is the investigation of tools to help automate the derivation of tight CBP parameters.

12 References

[1] 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, pp. 138–150, 1995.

[2] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software synthesis from dataflow graphs*, Kluwer academic publishers, 1996.

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

[4] 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.

[5] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, Jan. 1995.

[6] F. J. Harris. "Multirate FIR filters for interpolating and desampling." In *Handbook of Digital Signal Processing*. Academic Press, 1987.

[7] 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, 1990.

[8] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, Feb., 1987.

[9] P. K. Murthy, S. S. Bhattacharyya, "A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications," in *Proceedings of the International Symposium on System Synthesis*, November, 1999, to appear.

[10] P. K. Murthy, S. S. Bhattacharyya, *Shared Memory Implementations of Synchronous Dataflow Specifications using Lifetime Analysis Techniques*, Technical Report UMIACS-TR-99-32, University of Maryland at College Park, June 1999.

[11] P. Marwedel, G. Goossens, *Code Generation for embedded processors*, Kluwer Academic Publishers, 1995.

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

[13] 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*, October 1993.

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

[15] J. Vanhoof, I. Bolsens, H. De Man, "Compiling Multi-dimensional Data Streams into Distributed DSP ASIC Memory," Proceedings of the International Conference on Computer-Aided Design, 1991.

[16] 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*, pp. 193–200, 1991.