Look Into Details: The Benefits of Fine-Grain Streaming Buffer Analysis

Mohammad H. Foroozannejad Matin Hashemi Trevor L. Hodges Soheil Ghiasi

Department of Electrical and Computer Engineering
University of California, Davis, CA, USA
{mhforoozan,hashemi,tlhodges,ghiasi}@ucdavis.edu

Abstract

Many embedded applications demand processing of a seemingly endless stream of input data in realtime. Productive development of such applications is typically carried out by synthesizing software from high-level specifications, such as dataflow graphs. In this context, we study the problem of inter-actor buffer allocation, which is a critical step during compilation of streaming applications. We argue that fine-grain analysis of buffers' spatio-temporal characteristics , as opposed to conventional live range analysis, enables dramatic improvements in buffer sharing. Improved sharing translates to reduction of the compiled binary memory footprint, which is of prime concern in many embedded systems. We transform the buffer allocation problem to two-dimensional packing using complex polygons. We develop an evolutionary packing algorithm, which readily yields buffer allocations. Experimental results show an average of over 7X and 2X improvement in total buffer size, compared to baseline and conventional live range analysis schemes, respectively.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Algorithms, Performance

Keywords Streaming applications, Software synthesis, Synchronous Data Flow, Buffer management, Optimization

1. Introduction

Streaming applications are characterized by the need for processing a seemingly endless steady stream of input data as they are presented to the system. Typically, the processing demands access to a small window of input data and hence, the output can be generated and streamed out, as the input flows into the system. Streaming applications are abundant in the embedded and portable systems space. Examples include various encoding, decoding, transformation and inspection protocols in signal processing, multi-media, security and networking domains.

Most streaming application either exhibit fixed-rate behavior, or have fixed-rate kernels at the heart of the application [4]. Synchronous data flow graphs (SDF) [10] and its variations such as cyclo-SDFs [14], are widely used to model fixed-rate streaming applications. In these models, the functionality is specified as a number of independent tasks that communicate using channels with in-order data delivery. Among other purposes, the models are utilized to synthesize software implementation of the application.

Synthesizing embedded software from the models involves a number of challenges one of which, deals with implementation of inter-task communication channels [13]. The channels are often implemented as first-in-first-out (FIFO) buffers that are allocated as contiguous regions in the memory. Since streaming applications are data intensive, inter-task buffers tend to be large. As a result, buffers account for a substantial portion of the memory footprint of the synthesized programs.

In this paper, we study the problem of buffer management during synthesis of embedded software from SDF models. We argue that fine-grain perturbations to spatio-temporal behavior of individual buffers would enable aggressive sharing among buffers. That is, the time-dependent *details* of variations in buffered data should not be ignored. This is in contrast with conventional live range analysis techniques that do not allow overlap between two buffers if both of them happen to be *alive* at even one point in time.

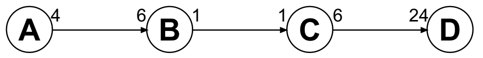
Following the *look into details* principle, we transform the buffer allocation problem into packing of complex polygons in the two dimensional time-space plane. We develop an evolutionary algorithm for the packing problem, which readily allocates buffers in the memory during compilation. The technique is implemented within the MIT StreamIt compiler, which compiles a specific variation of SDF. Experimental results on a number of applications show significant improvement in buffer size over existing competitors. The improvements are as high as 95.8% and 62.5%, and on average 85.9% and 52%, compared to the baseline and live range analysis-based approaches, respectively. This work complements our previous results on code generation for embedded multi-processors with limited memory [6].

2. Background

2.1 Application Model

Synchronous Data Flow (SDF) graphs are widely used to model streaming applications. Let V_G and E_G denote the set of vertices and directed edges of the SDF graph G, respectively. Vertices of the SDF graph, also known as actors, model application tasks, and directed edges represent inter-task communication channels. Edge e starts from the actor $src(e)$ (source), and ends at the actor $snk(e)$ (sink). Figure 1 depicts an example.

Upon execution, each task consumes a fixed number of data items, also known as tokens, from each of its input channels. The consumed tokens are processed to generate output data, which is subsequently written to output channels of the task after completion



V= { A, B, C, D}   E= { A_B, B_C, C_D }
S1: 6A 4B 4C 1D   Flat SAS
S2: 2(3A 2(1B 1C)) 1D   SAS
S3: 2A 1B 1C 4A 3(1B 1C) 1D   non-SAS
sink (A_B) = B   src (A_B) = A
cns (A_B) = 6   prod (A_B) = 4
q = [6, 4, 4, 1]   q[A] = 6   q[B] = 4

Figure 1. An example SDF graph, several valid schedules and some definitions are illustrated. SDF edges are annotated with corresponding production and consumption rates.

of the execution. The generated output also has fixed rate. Equivalently, each edge e is annotated with two $prod(e)$ and $cns(e)$ numbers, which refer to the number of tokens produced by $src(e)$ and consumed by $snk(e)$ upon execution, respectively.

Application tasks can be executed only after there are enough tokens to consume on their incoming edges. The produced tokens after execution of a task might enable execution of other tasks. Execution of a task is also referred to as firing of the corresponding actor in the model. Note that execution of a task implies that enough tokens already existed at its inputs. The streaming assumption implies that there are sufficiently-large number of tokens at the primary input, input from outside the model, to be processed.

2.2 Static Task Scheduling

Task can be executed in different orders, also known as task schedules. Due to production and consumption rates, task execution changes the storage requirement of the inter-actor channels. If repetitive execution of a fixed task schedule maintains the channels storage requirement bounded, the schedule can be utilized to synthesize an implementation at compile time. Such a schedule identifies one period of execution of the application, which is iteratively invoked to process the input stream.

It follows that a periodic task execution schedule has to meet two conditions: 1) actors can be fired only after there is enough tokens to consume on their incoming edges, 2) all of the generated tokens have to be consumed by the end of the period, to enable infinite repetition of the schedule using finite channel storage. It is well-known that realistic application SDF can be scheduled statically [9].

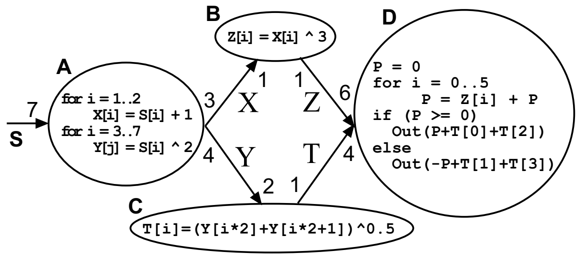
Let vector q denote the number of repetition of actors in the periodic schedule. Without loss of generality, we assume q refers to the simplest such vectors, i.e., not all of its elements can be divided by an integer larger than 1. To guarantee that all produced tokens are consumed by the end of the period, any static schedule has to guarantee the following for all edges of the SDF:

$$q[src(e)] \times prod(e) = q[sink(e)] \times cns(e)$$

The vector q is unique for real-life streaming applications [9]. Thus, the number of firings of actors in any static schedule is constant, although their ordering might differ in the period. In particular, "Single Appearance" (SA) schedule refers to the ordering, in which each actor appears exactly once. Figure 1 depicts an example SDF graph, along with several example schedules and notations.

2.3 Software Synthesis from SDF

To synthesize software from a given SDF model, one needs to determine a periodic ordering for execution of the tasks, which can be infinitely repeated. In the baseline synthesis scheme, task v appears in a loop whose iteration count is $q[v]$. Subsequently, the loops are "stitched" together in the given order, with appropriate



```
for i = 1..2
  X[i] = S[i] + 1
for i = 3..7
  Y[j] = S[i] ^ 2
```

```
Z[i] = X[i] ^ 3
```

```
T[i]=(Y[i*2]+Y[i*2+1])^0.5
```

```
P = 0
for i = 0..5
    P = Z[i] + P
if (P >= 0)
  Out(P+T[0]+T[2])
else
  Out(-P+T[1]+T[3])
```

Schedule: 2A 6B 4C D

```
while(1)
   for i = 0..1
     for j = 0..2
       X[i*3+j] = S[j] + 1
     for j = 3..6
       Y[i*4+j] = S[j] ^ 2
   for i = 0..5
     Z[i] = X[i] ^ 3
   for i = 0..3
     T[i]=(Y[i*2]+Y[i*2+1])^0.5

     P = 0
     for i = 0..5
       P = Z[i] + P
     if (P >= 0)
       Out(P+T[0]+T[2])
     else
       Out(-P+T[1]+T[3])
end While
```

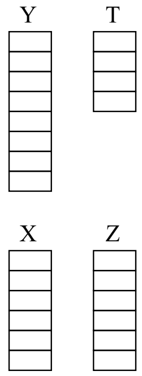


Figure 2. An example SDF, and the corresponding baseline implementation. Channels are implemented as distinct buffers.

fixtures to implement inter-task communication. Figure 2 illustrates the synthesized code for the depicted SDF.

SA task scheduling enables the synthesizer to save in application code size by instantiating tasks' internal computations exactly once, possibly within nested loops. The code size overhead of looping constructs is negligible with respect to typical size of task internal computations. Therefore, SA schedules are widely used in embedded systems, since they lead to small size synthesized software. In this work, we assume the given schedule to be SA, unless otherwise noted.

Recall that edge e in a SDF graph represents a FIFO communication channel between $src(e)$ and $sink(e)$. The channel stores the produced data after firings of $src(e)$, and its data is consumed during firings of $sink(e)$. Let $MT(e, S)$ denote the maximum number of tokens stored in channel e during firing of tasks according to schedule S. Clearly, $MT(e, S)$ indicates the minimum memory space required on this channel to implement the communication functionality.

The channels are typically implemented as buffer arrays to realize *in-order* communication with little cost. In the synthesized software, $src(e)$ writes into the buffer that implements channel e by maintaining a write index, referred to as the *Head*. The Head is reset at the beginning of the period, and is incremented after writing every token. The initial resetting enables reusing the same buffer memory in subsequent iterations. Similarly, $snk(e)$ maintains its own *Tail* index for reading from the buffer of channel e (buffer e for short), which is also reset at the beginning of the period, and is incremented after reading a token. Figure 2 illustrates the buffers in the synthesized code for the example SDF.

3. Buffer Memory Management

Streaming applications tend to require fairly large channel buffers, which is primarily due to the data intensive nature of their processing. As a result, total size of the buffer arrays usually accounts for a substantial portion of the application binary memory footprint. Enhanced management of the buffer memory can potentially lead to considerable reduction in memory requirement, which would be of great value in the resource constrained embedded space.

3.1 Baseline Buffer Allocation

For a given schedule S, the minimum size of the buffer e would be $MT(e, S)$. Smaller buffers would lead to incorrect or infeasible execution under S, because at least at one point during execution $MT(e, S)$ tokens need to be stored in the buffer e. In our discussions, therefore, we assume that the size of buffer e is exactly $MT(e, S)$.

The baseline synthesis scheme would be to allocate the buffers as independent regions in the data memory. In the "baseline buffer allocation" scheme, the buffers do not share any physical memory location at any point during execution. It follows that the overall buffer size would be the sum total of individual buffers, i.e., $\sum_{e \in E_G} MT(e, S)$. Figure 2 depicts a simple example.

3.2 The Impact of Scheduling

Changes to task scheduling can impact individual buffer sizes, which in turn, would influence total buffer memory requirement. In case of Figure 1, for example, $MT(A\_B, S_1) = 24$ and $MT(A\_B, S_2) = 12$. Note that under S_2, following production of 12 tokens by the actor A the consumer B gets fired, which consumes all of the tokens in the channel. Thus, the maximum number of tokens in the channel does not exceed 12. Unlike MT, the number of exchanged tokens over an edge does not depend on the schedule, and is only a function of the SDF structure and rates.

It is known that finding the optimal task schedule to minimize total buffer size is a NP-hard problem. Bhattacharyya et al. present two effective algorithms for constructing a single appearance schedule with emphasis on reducing the memory requirement [2]. Furthermore, phased scheduling has been proposed as a method for scheduling a SDF graph to minimize the memory size considering both code and data memory [8].

In addition to scheduling, the data memory requirement is impacted by the scheme used to allocate individual buffers in the memory. In this work, we direct our attention to this problem, i.e., minimizing overall buffer size through improved buffer analysis and allocation techniques. That is, we seek to improve buffer management without perturbing the given schedule.

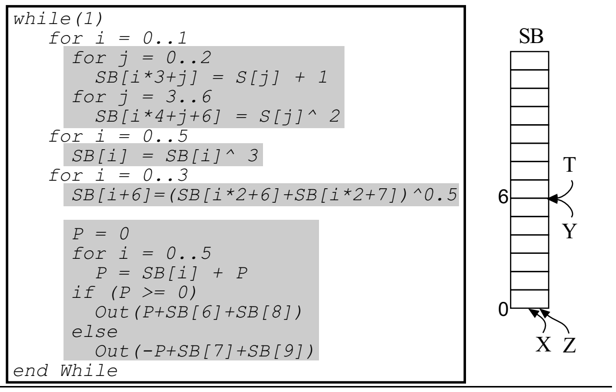
3.3 Buffer Sharing

In the baseline allocation scheme, separate portions of the memory are allocated to implement the channels of the SDF graph. During most of the execution time, however, the channel buffers are completely or partially empty. For example in figure 1, buffer A_B is completely empty during the firings of C and D in the first schedule. Therefore, the memory allocated to this buffer can be reused to implement buffer C_D. That is, the two buffers can *share* at least one physical memory location during execution, without compromising the functionality of the streaming application.

Figure 3 illustrates the synthesized code, under the buffer sharing assumption, for the example depicted in Figure 2. Note that buffers X, Y, Z and T are allocated at different offsets of the same array.

Extending the idea, any two channel buffers can be allocated to allow sharing of physical memory locations (space) as long as the two buffers do not conflict in time. In other words, the two buffers must not need to maintain a token at the same memory location at the same time.

The program synthesis framework, including its code generation protocol, impacts the possibility of sharing between two buffers. In this work, we assume that code generation follows the following rules:



```
while(1)
   for i = 0..1
     for j = 0..2
       SB[i*3+j] = S[j] + 1
     for j = 3..6
       SB[i*4+j+6] = S[j]^ 2
   for i = 0..5
     SB[i] = SB[i]^ 3
   for i = 0..3
     SB[i+6]=(SB[i*2+6]+SB[i*2+7])^0.5

     P = 0
     for i = 0..5
       P = SB[i] + P
     if (P >= 0)
       Out(P+SB[6]+SB[8])
     else
       Out(-P+SB[7]+SB[9])
end While
```

Figure 3. Shared buffer implementation of the SDF in Figure 2

1. None of the tokens of any buffer must be over-written or read by another buffer at anytime during the execution of the program.
2. Buffers must be statically allocated as contiguous regions in the application memory space.
3. The data should not be moved around within the buffer, i.e., data production and consumption operations are the only primitives to access FIFO channels. Token production and consumption increment Head and Tail indexes, respectively.

The rules collectively guarantee that the generated code conforms to SDF semantics, and the generated code safely implements the functionality. They eliminate the need for implementation of a complex inter-actor communication mechanism, which would incur large performance and code size penalty. Outstanding examples of academic and commercial SDF synthesis frameworks follow the same basic principles [1, 3].

4. Granularity in Buffer Analysis

The storage requirement (capacity) of any channel buffer changes with progress in the execution. The changes to the capacity of a buffer occur on execution of producer and consumer tasks of the corresponding channel. Such temporal change, however, can be captured at different levels of granularity [12].

The highest resolution temporal view of a buffer's storage requirement would need to follow the execution at the granularity of firing individual actors. In this scheme, execution of a task forms the unit of time for temporal analysis. We use the term *fine-grain* buffer analysis to refer to this level of abstraction.

At the other end of the spectrum, temporal changes to required capacity can be largely abstracted away. Specifically, buffers can be viewed to require storage of the maximum number of tokens during their lifetime. This leads to a *coarse-grain* temporal view of the buffer storage requirement, in which the buffer has the capacity $MT(e, S)$ during its live range, and zero otherwise.

A middle ground between the two ends of the granularity spectrum would be to consider consecutive execution of the same task as the time unit. In this level of abstraction, the storage requirement of the buffer is viewed to change only when a different actor is fired next. Among the consecutive firings of the same actor, the storage requirement of the buffer is viewed to be its maximum value in the range. Figure 4 illustrates a simple SDF, and the impact of granularity in temporal analysis of the buffer capacity.

4.1 Visualizing Buffer Analysis and Allocation

Figure 5 illustrates an example SDF, a given flat SAS, and three different buffer allocations. The allocations are visualized in a two-dimensional plane, in which the X-axis shows actor firings in the

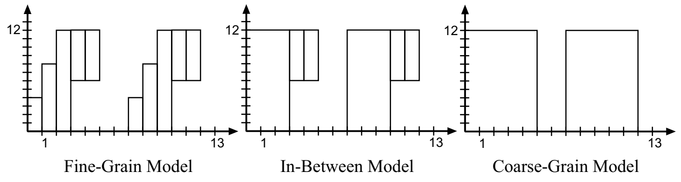


Figure 4. The impact of analysis granularity on the estimated temporal behavior of buffer A_B (Figure 1 under S_2). X and Y axis represent actor firings and buffer size, respectively.

schedule (time), and the Y-axis represents the buffer location in the memory (space). The unit of time is firing of a single actor.

The gray area of each buffer illustrates the range between Head and Tail indices that contain valid data. The temporal update in the gray area is due to the production and consumption operations, which increment the Head and Tail indices, respectively. The buffers are indexed relative to an *offset*. The offset indicates the start of the buffer, which is determined after the allocation process.

For a given analysis granularity, the capacity requirement of a buffer at any point in time is fixed. Thus, the X coordinate of any buffer in the two dimensional time-memory plane cannot be modified, i.e., the buffers can not be moved horizontally. The Y coordinate, however, represents the physical location of the allocated memory to implement the buffers.

The memory allocation problem can be viewed as a geometric layout instance, in which a solution is valid of the *laid out* buffers do not conflict in the time-memory plane. The only operation for perturbing the layout is vertical movement of the buffers. The geometric placement of a buffer in the plane readily gives its offset in the memory space. The objective is to minimize the vertical dimension of the layout, which represents the total size of the buffers.

4.2 Granularity and Buffer Allocation

The granularity in buffer analysis compromises accuracy in temporal behavior of buffers with analysis complexity. Our conviction is that fine-grain analysis, though more expensive in terms of analysis complexity, provides temporal details that enable *substantial* improvements in buffer sharing. The three layouts in the Figure 5 illustrate the idea.

Figure 5.A shows the baseline buffer allocation scheme, in which every buffer is assumed to have maximum capacity throughout the execution. Therefore, buffers have to be allocated in dedicated locations in the memory. The figure shows that the total size of channel buffers is 45.

Figure 5.B depicts the optimal buffer allocations under coarse-grain analysis of their temporal behavior. In this scheme, buffers are assumed to have maximum capacity throughout their live range in the schedule. Therefore, two buffers would conflict if they are alive in at least one point in time in which case, they cannot share any physical memory location and have to be allocated in distinct memory spaces [12].

For example $MT(A\_B, S) = 6$, and under the coarse-grain analysis model six memory cells have to be allocated during its entire life time (three time steps) to implement this buffer. The gray areas in the picture illustrate the actual storage requirement, and the border lines represent the memory assignments for each buffer. The white space inside the borders represent memory space that is left unused at the corresponding point during execution. In the example of figure 5.B, the total size of channel buffers is 24.

Finally, figure 5.C shows the optimal allocation of buffers under fine-grain analysis scheme, in which, buffers' temporal behavior is updated at the granularity of actor firings. Intuitively, fine-grain view of the buffers' spatiotemporal patterns enables more

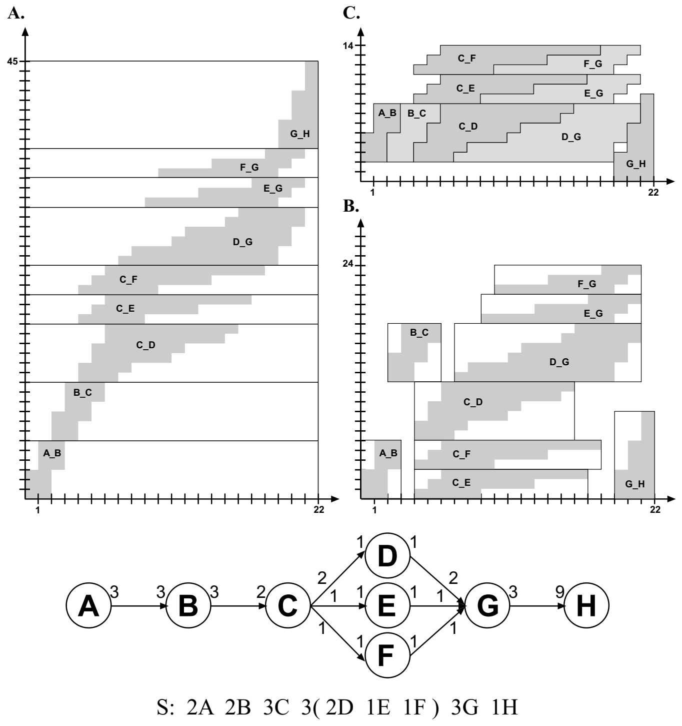


Figure 5. A. Baseline, **B.** Coarse-grain, and **C.** Fine-grain allocation schemes for the illustrated example. X and Y axis show actor firings in the schedule, and the offset within the memory space, respectively.

condensed packing of the buffers in the memory, which translates into smaller code size. In this example, the total size of channel buffers is 14.

5. Fine-Grain Buffer Allocation

The temporal behavior of FIFO buffers can be characterized as a pair of two Head and taiL vectors. H keeps the head index $H_e[t]$ and L keeps the tail index $L_e[t]$ at the time step t of the program. The concept of time here is the same as that we described in 3, therefore the length of H and L is the maximum time steps in one iteration of the program which is the summation of the elements in q_G.

$$\forall e \in E \;:\; B_e = (H_e, L_e)$$

B_e : The Buffer on edge e which we call it buffer e in short

$H_e[t]$: Head index at time $0 \leq t \leq T$ for the buffer on e

$L_e[t]$: Tail index at time $0 \leq t \leq T$ for the buffer on e

$$T = \sum_{v \in q_G} q[v]$$

In this definition of buffers head and tail indices start from zero and go up to their maximum level and go back to zero again after getting to the end. In the notion of buffer sharing, each buffer is allocated within the shared buffer which means an offset will be assigned to each buffer and will be added to the head and tail of the buffer in a way that no conflict would occur. Note that H and L keep the value of head and tail for each individual buffer without considering the sharing scheme. The offset which will be assigned to each buffer indicates the true location of them within shared buffer. Assume tuple O keeps the offsets for all the buffers on all the edges of the graph:

$$O = \{(o_{e_1}, o_{e_2}, o_{e_3}, \ldots, o_{e_N}) \mid e_1 : e_N \in E\ ,\ N = |E|\}$$

o_e is the offset for the buffer on edge e

The following is the definition of SBS as "Shared-Buffer Size" and the objective of the problem is to minimize SBS:

$$SBS = \max_{\forall e \in E}\{o_e + H_e^{max} \mid H_e^{max} = \max_{0 \le t \le T}(H_e[t])\}$$

The following lemma gives us one of the advantages of using SA scheduling and will help us to specify the constraints of the problem:

LEMMA 5.1. *In SA schedules the head index at the time t is always greater than equal the tail index at the same time:*

$$\forall t \le T\ :\ H_e[t] \ge L_e[t]$$

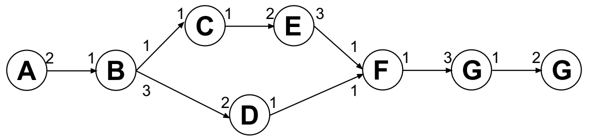
Proof: The opposite situation might occur when there have been already some tokens written in and read from the buffer, thus both head and tail are pointing to a place in the middle of the buffer. If there are more tokens to be written in the buffer and head has already hit the end, it will start from the beginning of the queue to fill up the empty spots (head never surpasses tail because we assume the size of the buffer is big enough to keep all the required tokens). In this situation, head will be pointing to a location in the vicinity of zero while tail is still somewhere around the maximum size of the buffer.

The claim is obvious for a flat SA schedule. The producer gets the chance to fill up the buffer and it is when head starts going up to its maximum position and stays there. Later on the consumer comes in and empties the buffer and now this is the turn for the tail index to go up. Eventually tail meets head at the end point of the buffer and they both go to zero. Since both the producer and the consumer appear only once in the schedule without being repeated in nesting loops, head and tail never change again in the current iteration of the program. The next iteration will start with an empty buffer exactly the same as the first iteration. However when we have nesting loops, both producer and consumer might be repeatedly fired one after each other.

Let the $\underline{R}ound$ of the edge e and schedule S which we denote $R(e, S)$ be a sequence of actors in schedule s from when $src(e)$ starts being fired to when $sink(e)$ is fired and we are back to $src(e)$ again. This sequence includes the direct repetitions of the both source and sink actors. Because S is a SA schedule, every time $src(e)$ is fired, the same path will be taken to $sink(e)$. In fact if we replace this sequence of actors with the notation $R(e, S)$ in our schedule, none of the $src(e)$ nor $sink(e)$ will appear alone out of $R(e, S)$ in the schedule.

Figure 6 depict an example of this replacement for the given edge and schedule. Therefore all the tokens produced on the edge e during $R(e, S)$ should be consumed by the end of each round, otherwise they will be accumulated at the end of the schedule which is against the definition of a valid schedule. Therefore we only need to prove the claim within a round, which is true because based on the definition of a round, $src(e)$ appears only in the first portion of the round and does not appear again after $sink(e)$ is fired.■

In the problem of buffer sharing, the vectors H and L are known and given. We also have a consistent SDF graph and a valid SA schedule which means at any given time we know the number of tokens exist on each edge of the graph and the relative value of head and tail. Therefore lemma 5.1 does not impose any constraint



S: 6A 2(3(2(B C) 1E 3(D F)) 3G) 3H
R(B_D, S) = B C B C E D F D F D F
New Sequence with R: 6A 3R 3G 3R 3G 3H

Figure 6. SA schedule with nested loops and the notion of "Round". The example is for buffer B_D

to the problem. However by using this lemma we can specify the constraint and the objective of the problem as the following:
Constraint:

$$\forall\, e, b \in E \quad \forall\, 0 \le t \le T\ :$$
$$H_e[t] + o_e \le L_b[t] + o_b \quad OR \quad H_b[t] + o_b \le L_e[t] + o_e$$

Objective:

$$minimize \quad SBS$$

The constraint suggest that no buffer can write in or read from the significant data of another buffer. Therefore for buffer e at any given time the meaningful data of other buffers have to be assigned after or before this buffer. The real location of head and tail within the shared buffer is relative to the assigned offsets (o_e and o_b). Since these offsets will not change in time, thus a place in shared-buffer is assigned to the buffer only once, and the data will be written in and read from this location in the entire program without conflicting with any other buffer nor requiring any data to be relocated.

6. ILP Formulation

Integer Linear Programming (ILP) provides a mechanism to obtain the optimal solution of a problem as long as its constraints and objective can be described as linear constraints of integer variables. Since there are many commercial ILP solvers available, one only has to cast the problem in ILP formulation to solve a specific instance. In case of the buffer sharing problem, linear constraints have to ensure that all buffers are allocated without any conflict.

The subtle difficulty in such formulation is to avoid buffer conflicts using linear constraints, because two conflicting buffers can be allocated in either order in the shared buffer. In other words, formulation of the "OR" logic is non-trivial, since a buffer can be allocated either before or after another conflicting buffer as long as there is no violations of the stated guidelines.

Because linear constraints cannot be easily used to articulate the "OR" logic, we had to reformulate the problem. For each buffer and each location in the shared memory space, specifically, we define a binary variables, whose '1' value would indicate allocation of the corresponding buffer in the corresponding memory location. Subsequently, buffer conflict constraints can be formulated as a large number of linear constraints. Note that the constraints have to be generated for all time steps. We do not include the details of the formulation for brevity.

The complexity of buffer sharing instance, and ILP runtime grows exponentially. Therefore, it does not provide a scalable approach to solving the buffer sharing problem. Nevertheless, we utilize ILP to obtain the optimal solution to problem instances, although at the cost of unreasonably long solver runtime, primarily for measurement of the optimality gap using other techniques (Section 10).

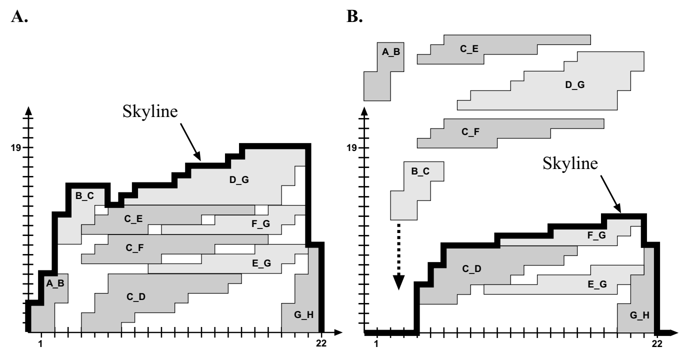


Figure 7. For the SDF graph of Figure 5: **A.** A non-optimal permutation: (G_H, C_D, A_B, E_G, C_F, F_G, C_E, D_G, B_C). The line in bold is the final skyline. **B.** MDA is moving down the buffers in following order: (G_H, E_G, C_D, F_G, B_C, C_F, D_G, A_B, C_E). Each step forms a skyline with the placed buffers.

7. Strip Packing Problem and Buffer-Sharing

In several industries there is a need for packing a set of 2-dimensional objects on a larger rectangular unit of material by minimizing the waste. This larger unit can be a standardized sheet of material, from which the set of objects have to be cut. The objective is to pack all the items into the minimum number of units. This problem is a variation of the well-known bin-packing (BP) problem, and is used in some industrial applications such as wood or glass industries.

In other contexts the standardized unit is a roll of material such as a roll of paper or cloth and the objective is to use the minimum roll length. This problem is called strip-packing (SP) problem and we will be using it in this paper to realize the buffer-sharing problem. These two problems are known as NP-complete and there has been various attempts to solve them in the algorithms community. [11] provides a survey on some of the two-dimensional packing problems and solutions.

In the context of buffer sharing one can realize a large array of memory (which we call shared-buffer) analogous to the roll of material in the SP problems, and the different buffers on different edges of the graph could be the set of objects. Figure 5 shows the geometrical aspect of buffers where we have time on one axis and the indices of shared-buffer on the other axis. In this model the objects are being constructed from the number of tokens that exist in the buffer during the run time of the program. Subsequently, we adopt a SP packing algorithm proposed in [7] with some adjustments specific to the buffer-sharing problem.

8. Move-Down Algorithm

Move Down Algorithm (MDA) is the main piece of our method and will be most repeated during the run time of the main algorithm. It also gives us the ability to make sequences among our input which is essential for Genetic Algorithms. The idea is very simple. we push each buffer toward the beginning of the shared-buffer array as much as possible so that they will take less space after all being allocated. As we can see in figure 5 the order of moving down the buffers matters and some of the sequences take less space than others. Figure 7A. shows another sequence of buffers from the SDF graph of figure 5 which is not optimal.

To understand how far a buffer can go down we introduce another vector which is called skyline and denoted $Vsk[t]$. In this section we consider buffers as solid geometrical shapes which can stand on the top of each other to construct a wall. Looking this way, skyline is the highest level of the constructed wall in each time step. Figure 7 shows two different skylines for different situations while running MDA.

To construct the skyline vector we introduce skyline function which takes a Vsk and buffer B_e and also an offset o to place the buffer, and it will calculate the skyline vector $\acute{V}sk$ constructed from adding the new buffer at the point o to the existing skyline.

$$\forall e \in E \quad \forall 0 \leq o \leq BBS \quad \forall 0 \leq t < T \ :$$
$$\acute{V}sk = skyline(Vsk, e, o) = \begin{cases} Vsk & \text{if } S_e[t] = 0 \\ h_e[t] + o & \text{otherwise} \end{cases}$$

MDA takes a skyline vector and a buffer, and returns the lowest offset it can get from pushing down this buffer before hitting the skyline.

$$o_e = MDA(Vsk, B_e)$$

The algorithm first places the buffer on the first level of the skyline by setting the offset to $Vsk[0]$. Then moving to the right, it compares the skyline to the seated buffer to see if there is any conflict and if there is one, it will adjust the offset to remove the conflict. Move Down Algorithm:

```
$o_e := Vsk[0]$
$for\ \ i := 0 \rightarrow T$ :
    $if(S_e[t] \neq 0)\ and\ (H_e[t] - S_e[t] + o_e < Vsk[t])$ :
        $o_e := Vsk[t] - H_e[t] + S_e[t]$
$return\ \ o_e$
```

If we run MDA on all the buffers in a pre-defined order, and also calculate the skyline on each step and use it for the next step, we have done the buffer allocation. If we call the pre-defined order a permutation of buffers and denote it with π, then we have:

$$O = PlaceAll(\pi)$$
$$\pi = (B_1, B_2, B_3, \ldots, B_N)(\text{any order of buffres})$$
$$O = (o_{e1}, o_{e2}, o_{e3}, \ldots, o_{eN})$$

The function $PlaceAll$ will place each buffer in the same order they have in π as follows:

$$o_{e1} = MDA(Vsk_1, B_1) = 0$$
$$o_{e2} = MDA(Vsk_2, B_2)$$
$$\ldots$$
$$o_{eN} = MDA(Vsk_N, B_N)$$

The first buffer always gets zero for the offset, and it is because we are pushing down the buffers as much as possible and there is nothing in the shared-buffer yet so it goes all the way down. For the skyline vectors we have:

$$Vsk_1 = V_{zero}(\text{the vector zero})$$
$$Vsk_2 = skyline(V_{zero}, B_1, 0) = H_{e1}$$
$$Vsk_3 = skyline(Vsk_2, B_2, o_{e2})$$
$$\ldots$$
$$Vsk_N = skyline(Vsk_{N-1}, B_{N-1}, o_{en-1})$$
$$Vsk_{N+1} = skyline(Vsk_N, B_N, o_{eN})$$

The very first skyline vector is $V_{zero} = [0, 0, \ldots, 0]$ which we can consider the ground. The second skyline forms when we push the first buffer down to the ground. Therefore skyline forms exactly on the top of this buffer which is the vector H.

The final skyline is Vsk_{N+1} and determines the height of the wall which is actually the size of shared-buffer:

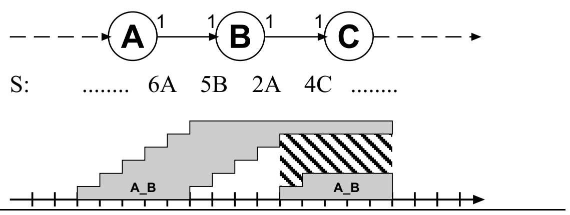


Figure 8. Part of a SDF graph with a non-SA schedule. The figure shows that for the buffer A_B the striped space is not reachable by MDA.

$$SBS = \max_{0 \le t \le T} (Vsk_{N+1}[t])$$

The size of shared-buffer depends on the sequence of buffers we are using, and $PlaceAll$ itself does not guarantee that it will give us the optimal solution. However because it uses the notion of sequence in placing the buffers, it reduces the search space from $(BBS)^N$ (all the possible places for buffers to be in an array with the size of BBS) to $N!$ (the number of different sequence of buffers that we can have). Moreover having a sequence of data as the input, is one of the fundamentals of a genetic algorithm and enables us to use them to find the optimal or near optimal solution.

Lastly in this section, the following lemma shows weather MDA is capable of giving us the optimal solution or not. In fact this is another advantage of using SAS:

LEMMA 8.1. *The optimal solution can be found by using MDA in SA schedules. It is only the matter of finding the right sequence of buffers.*

Proof: Figure 8 depicts a buffer in a non-SA Schedule and as we can see there is an area inside buffer A_B which MDA can not reach. However because of lemma 5.1, in a SA schedule the head index is always greater than the tail index, thus any empty space inside a buffer is open from the top or the bottom of the buffer and can be reached by MDA.■

9. Evolutionary Optimization using MDA

In this section, we utilize the move-down algorithm to construct an evolutionary genetic optimization technique. Genetic optimization is composed of several key components, including chromosome, inheritance and fitness function. Chromosome provides an abstract representation of solutions in the search space, and is normally represented as a sequence of numbers. Inheritance models the basic operations through which, chromosomes are perturbed to improve the solution quality. Typically, there are two crossover and mutation inheritance operations in a genetic optimization framework. Finally, the fitness function quantifies the "quality" of candidate solutions, and determines survival of selected candidates. Our objective is to define the notions of chromosome, inheritance and fitness, in the context of buffer sharing, and subsequently, utilize genetic optimization to solve our problem at hand.

MDA provides the ability to work on a sequence of buffers as the input and to allocate all of them inside the shared buffer according to their order in the sequence (Section 8). The size of the shared-buffer is the height of the final structure, in the corresponding packing instance. We propose to use different permutations of buffers as chromosomes, or individuals of a population, and the height of the final skyline as the fitness function, in the genetic optimization framework. Consequently, the algorithm will work in the following steps:

To initialize the algorithm with a sample population, we randomly select a set of permutations. The size of the sample population is a pre-defined parameter. We used the number of buffers ($N = |E|$) to be the size of the population in our algorithm.

$$\text{Sample set} = \{\pi_1, \pi_2, \pi_3, \ldots, \pi_N\}$$

Since genetic algorithm keeps track of different lines of breeding patterns, having a larger sample population gives us the ability to keep track of more candidate solutions. On the other hand, having a very large population slows down the algorithm, and reduces the chance of finding the optimal solution in a reasonable time.

Now we can run $PlaceAll$ algorithm and calculate the height of the final solution in every individual permutation in the set. We choose the height of each permutation (denoted as $height(\pi)$) to be the fitness function (denoted as $f(\pi)$) as follows:

$$f(\pi) = \frac{1}{height(\pi)}$$

For any permutation there is a chance that part of its sequence matches the sequence in the optimal solution. Basically, we would like to find these parts from different members and concatenate them, so that we can get closer to the optimum. The mechanism to recognize if we are getting closer to this goal is the fitness function. To generate new members first we need to select two of the existing members, which we refer to as parents. We select the parents depending on their fitness. The fitter individuals (shorter in height), have a higher chance of being selected. The probability of selection of an individual permutation (denoted as $p(\pi)$ is likely to change in each iteration of the algorithm due to changes to the fitness of the other members of the group.

$$p(\pi_i) = \frac{f(\pi_i)}{\sum_{j=1}^{N} f(\pi_j)}$$

In practice we can divide the interval $[0, 1)$ into N sub-intervals as follows:

$$[0, p(\pi_1)) \ , \ [p(\pi_1), p(\pi_2)) \ , \ \ldots \ , \ [p(\pi_{N-1}), p(\pi_N))$$

Two random numbers from the interval $[0, 1)$ will determine the selected permutations.

Subsequently, the parent chromosomes are used to create the children using the crossover operation. Our crossover function generates two random numbers $1 \le p \le q \le N$. Then it copies the sub-sequence of the first parent from position p to q, and place it at the beginning of the child's chromosome. The sequence from p to q is the part that we would like to preserve, hoping that the same sequence exists in the optimal solution. Finally, we fill the rest of the offspring with the remaining genes (buffers) in the second parent in the same order that they appear in the second parent. The following example shows how crossover function works:

$$\begin{aligned}
&p = 2 \quad q = 4 \\
&\pi_{parent1} = (B_{e1}, \underbrace{\mathbf{B}_{e2}, \mathbf{B}_{e3}, \mathbf{B}_{e4}}, B_{e5}, B_{e6}) \\
&\pi_{parent2} = (\mathbf{B}_{e6}, \mathbf{B}_{e5}, B_{e4}, B_{e3}, B_{e2}, \mathbf{B}_{e1}) \\
&\pi_{child} = (B_{e2}, B_{e3}, B_{e4}, B_{e6}, B_{e5}, B_{e1})
\end{aligned}$$

Copying and matching different sequences from existing permutation may lead the process to stay in a local minimum region. To avoid this situation we can mutate the child based on the probability $p_{mutation}$, which is another parameter of the algorithm. If the child is to be mutated, then the function generates two random numbers $1 \le i, j \le N$, and swaps the buffers in those positions within the sequence.

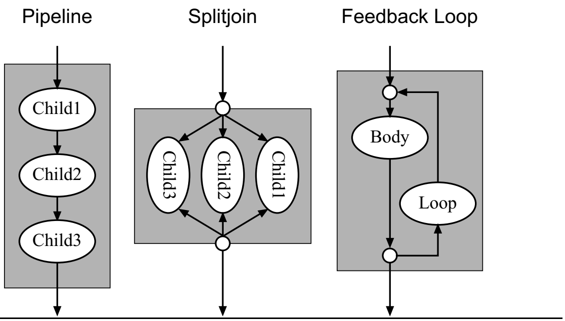


Figure 9. The composite objects of StreamIt language

$p_{mutation} = 0.4$: the probability of being mutated

$i = 2 \quad j = 4$

π_{child} Before $= (B_{e2}, \mathbf{B}_{e3}, B_{e4}, \mathbf{B}_{e6}, B_{e5}, B_{e1})$

π_{child} After $= (B_{e2}, B_{e6}, B_{e4}, B_{e3}, B_{e5}, B_{e1})$

The $PlaceAll$ algorithm is run on the newly generated child to calculate the height of the offspring. The child is then added to the population set. To maintain the pre-defined population of the sample set we kill (remove) the weakest (highest) member of the sample set. Therefore the offspring will be compared against the weakest member of the population, and may or may not remain in the sample set.

Iteratively, we generate new children and compare them to the existing members until the termination point where we can return the best solution found. Termination can be an acceptable size of the shared buffer (the height of the best permutation). Alternatively, the optimization can be terminated at a time limit. We selected the number of iterations as the termination criterion. We set the value to be the product of N and an iteration parameter.

10. Experimental Evaluation

10.1 Setup

We have integrated our algorithm into the MIT StreamIt compiler [5]. StreamIt refers to both a programming language developed for specifying the streaming applications, and a java-based compiler. The StreamIt language conforms to the SDF semantics, by modeling an application as a graph of interconnected but independent “filters” with statically-defined input and output rates that communicate via FIFO channels.

StreamIt utilizes four stream objects to hierarchically build the application graph: *filters* form the basic data processing unit, while the other three objects, *pipeline*, *split-join*, and *feedback loop* (Figure 9), are composite objects that contain children stream objects. The children are recursively constructed out of the four different object types. In other words, the three composite stream objects (*pipeline*, *split-join*, and *feedback loop*) act as containers to build different graph structures, and the *filter* specifies data processing. The design ensures that the graph is highly structured while providing the programmers with a simple, yet flexible, set of objects to construct the stream graph for their streaming application.

We used the built-in single appearance scheduler to construct task execution order. The StreamIt scheduler is designed based on the hierarchical nature of the language. Specifically, the composite objects form virtual tasks, under which its children are scheduled. If any of the children happens to be a composite object itself, its firing in the schedule will be replaced by its own children. Consequently, only filters appear in the schedule as data processing actors. Figure 10 depicts a simple example.

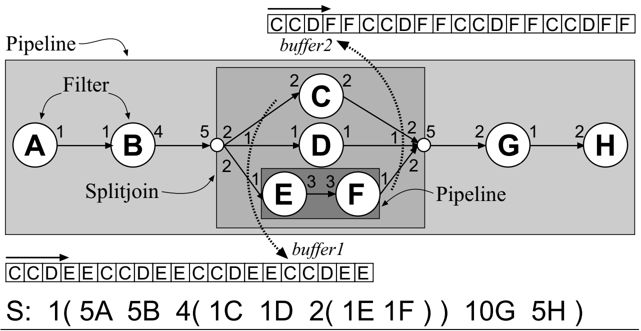

C C D F F C C D F F C C D F F C C D F F
buffer2
Pipeline
Filter
A
B
C
D
E
F
G
H
Splitjoin
Pipeline
buffer1
C C D E E C C D E E C C D E E C C D E E
S: 1(5A 5B 4(1C 1D 2(1E 1F)) 10G 5H)

Figure 10. The split-joins share a buffer at their start and end terminals. The split-join in this picture works in round robin fashion. The letters inside *buffer1* and *buffer2* illustrate the mapping of array elements to the actors.

	Number of Buffers	Number of Actores	Number of Time Steps	Base-line	Coarse-Grain	Fine-Grain (Best Case)	Fine-Grain (Worst Case)	Compile Time with GA in Sec.	Optimal Solution by ILP
Bitonic Sort	119	214	340	1152	96	48	64	91	32
Insertion Sort	8	9	263	1024	256	128	128	6	128
FFT2	22	24	446	3072	640	384	384	10	~
FFT3	38	64	175	960	192	72	96	11	64
TDE	48	51	17204	77120	23168	11776	23040	510	~
Matrix Mult.	10	21	2712	5000	4000	2000	2000	13	~

Figure 11. Benchmark characteristics and experimental results

The StreamIt compiler translates stream programs to C, which can be passed to any standard C compiler to generate executable binaries. The compiler defaults to the baseline buffer allocation scheme, in which channels in the stream graph are implemented with distinct arrays. We instrumented the compiler to allocate all the buffers within the same array, though at different indices. The baseline and instrumented synthesized codes were compiled and executed on a Unix machine to ensure that functional correctness is preserved after our transformation.

10.1.1 Benchmark Applications

We selected six different streaming kernels as our benchmarks to evaluate the proposed technique. They include two sorting algorithms, two different implementation of the fast Fourier transform (FFT), time delay estimation (TDE) and matrix multiplication kernels. These kernels frequently appear in many higher-level application that are used in portable and handheld embedded systems.

The table in Figure 11 shows the benchmarks. The benchmarks are implemented in the StreamIt language. The second and third column of the table list the complexity of each application in terms of number of channel buffers and actors (tasks), respectively. The fourth column of the table shows the number of time steps, i.e. sum total of task executions in the periodic schedule, for each benchmark.

Note that unlike generic SDF tasks, StreamIt filters have only one input and one output buffer. More complex inter-actor communications are modeled using split-join objects. In synthesizing split-joins, one large buffer is used to implement multiple channels that either split to or join from several actors. The sinks of a split (or sources of a join) read from (write to) the corresponding locations in the large buffer (Figure 10). The size of the large buffer is the sum total of the individual channel buffers, i.e., no sharing between channels is performed when allocating them in the same

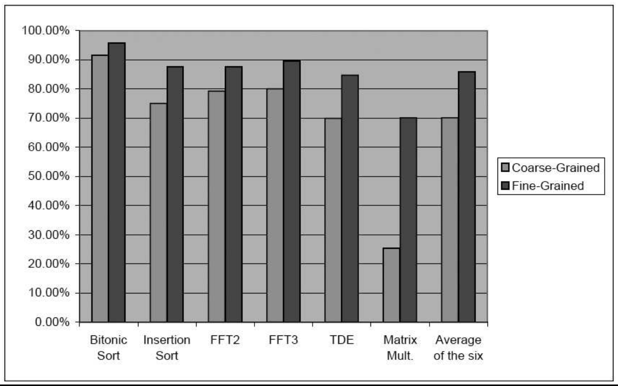


Figure 12. The Improvement of coarse-grain and fine-grain methods compared to the baseline.

buffer. Consequently, the number of buffers in the StreamIt program tends to be less than the number of actors. Figure 10 depicts a split-join and the buffers in its terminals.

In general, stream programs might have two different initialization and steady state execution phases. The initialization phase might be needed if a non-trivial buffer content configuration has to be created to enter the steady state phase. In this work, we have focused on the steady state buffer analysis of the stream programs.

10.1.2 System and Algorithm Configuration

Our proposed genetic algorithm for buffer optimization uses a number of configurable parameters. In our experiments, we have set the iteration number of the algorithm to $1000\times$ number of buffers in the application. In addition, the sample population in the genetic algorithm is configured to be equal to the number of buffers in the application, and the probably of mutation operation is 0.4. The parameters are configured with very small effort to create a reasonable balance between optimization runtime and solution quality. The experiments are performed on a Unix PC with Intel Pentium 4 CPU running at 2.80GHz, 1024KB of cache, and 3GB of main memory.

10.2 Results

Figure 11 shows the result of baseline, coarse-grain and our genetic-algorithm based fine-grain buffer allocation on the benchmark applications. The genetic algorithm is intrinsically non-deterministic. We report the worst and the best buffer size that we observed in 10 runs.

The coarse-grain analysis is done according to the buffer lifetime analysis principle, developed by Murthy and Bhattacharyya [12]. The *first fit* heuristic is used to allocate the buffers in the shared buffer, under the same SA schedule. Note that first fit algorithm is concluded to perform well under coarse-grain buffer analysis model [12].

We also generated ILP instances for the benchmark application, according to the formulation developed in Section 6. Due to the exponential growth of the ILP complexity with respect to time steps for the problem, our system was unable to load the enormous-sized ILP instances of *FFT2*, *TDE*, and *Matrix Mult*. The ILP approach is clearly not scalable, nevertheless, it is helpful in establishing an optimality gap for the selected benchmarks.

Figure 12 visualizes the performance of coarse grain and fine grain buffer allocation, in terms of savings in total buffer size over the baseline scheme. Our evolutionary allocation method reduces the buffer size by 86% on average. The improvements are as dramatic as 24X in case of *Bitonic Sort*.

Figure 13 compares the performance of fine-grain analysis, either via using ILP or our proposed algorithm, versus the conventional live range analysis coarse-grain method. The data shows that both the best and worst case results of our scheme outperform the coarse-grain results. Considering the best case, our allocator improves the result of coarse-grain allocation by about 52%, and is only marginally inferior to the theoretically optimal ILP solution. Note that the best and worst case results are collected over only 10 different runs. Thus, it is reasonable to consider best case performance for comparison purposes, given our algorithms' decent latency and the small number of required repetitions.

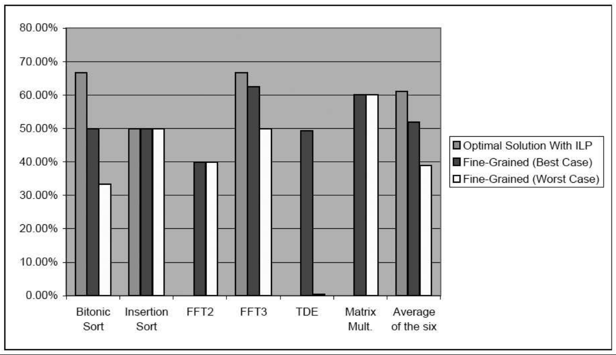


Figure 13. The Improvement in all fine-grain cases: GA worst case, GA best case, and ILP, compared to the coarse-grain method.

The complete compile time in our experiments varied from a few seconds to a few minutes depending on the complexity of the program. We believe that the compilation latencies are quite reasonable, considering their absolute value, our old experimentation platform, and the fact that compile-time analysis latency is justified given the gains in application memory footprint.

Varying the analysis granularity involves a natural tradeoff between buffer size and optimization latency (compiler runtime). It is important to strike a balance between the two competing elements. Our study has showed that not only fine-grain analysis is not to be dismissed due to its complexity [12], but it can be the method of choice in a large number of application scenarios.

11. Conclusions

Streaming kernels and applications are abundant in the embedded systems domain, where underlying hardware platforms have to deal with strict resource constraints. It is critical to optimize the streaming applications for resource requirement, such as their memory footprint. We contribute to this important goal by developing a novel buffer allocation technique during synthesis of streaming applications from synchronous dataflow specifications.
We argue that fine-grain analysis of buffers' temporal behavior, as opposed to conventional coarse-grain live range analysis, enables dramatic improvements in buffer sharing. We transform the buffer allocation problem into packing of complex polygons in the two-dimensional space, and present an evolutionary algorithm to solve the problem. Experimental results demonstrate the superiority of our approach compared to existing competitors in terms of the memory footprint of the synthesized applications. We conclude that the benefits of considering buffers' fine-grain temporal behavior outweighs the reasonable increase in static analysis latency for a large class of resource-constrained embedded systems.

Acknowledgments

The support of National Science Foundation, under grant CCF-0903549, and Semiconductor Research Corporation, under contract 2009-HJ-1971, is gratefully acknowledged.

References

[1] Mathworks simulink - simulation and model-based design. available online at http://www.mathworks.com/products/simulink/.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[4] M. Geilen and T. Basten. Reactive process networks. In *International Conference on Embedded Software*, pages 137–146, 2004.

[5] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. P. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[6] M. Hashemi and S. Ghiasi. Versatile task assignment for heterogeneous soft dual-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2010.

[7] S. Jakobs. On the genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88:165–181, 1996.

[8] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Conference on Languages, Compilers and Tools for Embedded Systems*, pages 103–112, 2003.

[9] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), 1987.

[10] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[11] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: a survey. *European Journal of Operational Research*, 141 (2):241–252, 2003.

[12] 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*, 20(2), 2001.

[13] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[14] M. Wiggers, M. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Design Automation Conference*, pages 658–663, 2007.