Static Memory Management for Efficient Mobile Sensing Applications

Farley Lai, Daniel Schmidt, Octav Chipara
Department of Computer Science
University of Iowa
{*poyuan-lai, daniel-schmidt, octav-chipara*}@uiowa.edu

ABSTRACT

Memory management is a crucial aspect of mobile sensing applications that must process high-rate data streams in an energy-efficient manner. Our work is done in the context of synchronous data-flow models in which applications are implemented as a graph of components that exchange data at fixed and known rates over FIFO channels. In this paper, we show that it is feasible to leverage the restricted semantics of synchronous data-flow models to optimize memory management. Our memory optimization approach includes two components: (1) We use abstract interpretation to analyze the complete memory behavior of a mobile sensing application and identify data sharing opportunities across components according to the live ranges of exchanged samples. Experiments indicate that the static analysis is precise for a majority of considered stream applications whose control logic does not depend on input data. (2) We propose novel heuristics for memory allocation that leverage the graph structure of applications to optimize data exchanges between application components to achieve not only significantly lower memory footprints but also increased stream processing throughput. We incorporate code generation techniques that transform a stream program into efficient C code. The memory optimizations are implemented as a new compiler for the StreamIt programming language. Experiments show that our memory optimizations reduce memory footprint by as much as 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled. These results suggest that highly efficient stream processing engines may be built using synchronous data-flow languages.

1. INTRODUCTION

The advancing capabilities of smartphones have enabled the development of a new generation of mobile sensing applications such as those for context monitoring [1], user identification [2], personal health [3], and environmental monitoring [4]. At the heart of these applications, there are sophisticated stream engines that must process high-rate sensor data efficiently.

Memory management is a key challenge in the development of stream engines. Previous studies have shown that poor memory management leads to applications with large memory footprints and excessive memory accesses that reduce stream processing rates [5, 6, 7]. A common source of inefficiency is the fact that stream operations such as windowing, splitting, appending, and downsampling have been traditionally implemented using memory copying. Avoiding copying requires the memory management to support data sharing among the components of an application. Data sharing typically reduces both memory usage and the number of memory accesses. However, in order to improve the stream processing rate, we must ensure that data sharing does not reduce cache locality or increase code complexity.

The problem of effective memory management for stream processing may be addressed through dynamic or static memory management. Dynamic memory management relies on specialized data structures for manipulating streams. A representative example is the SigSeg [6, 8]. A SigSeg is organized as a list of buffers containing data samples; each buffer may be shared between components using reference counters. While the SigSeg is a clever data structure, dynamic memory management suffers from two intrinsic limitations. (1) Dynamic memory management can exploit only a fraction of data reuse opportunities as some run-time overhead may be introduced for analyzing an application's behavior. (2) The data structure unavoidably adds a level of indirection in accessing streams of samples, which reduces the performance of stream engines.

Compile-time solutions may be used to determine memory allocations without introducing run-time overhead. The key is to develop a static analysis technique that may precisely identify *location* and *temporal sharing* opportunities. Unfortunately, static analysis of general purpose languages such as C or Java quickly becomes imprecise because of complex control structures and pointer aliasing. We avoid these difficulties by focusing on a domain-specific stream processing language called StreamIt [9]. We show that it is feasible to leverage the constrained semantics of stream programs to implement stream operations efficiently through static memory allocation. Since StreamIt is a representative example of a synchronous data-flow (SDF) language, we expect that the results presented in this paper will translate into other systems and languages based on SDF models.

We propose Efficient Static Memory management for Streaming (ESMS) that addresses the above challenges. In

.

978-1-4673-8079-9/15/$31.00 ©2015 IEEE

this paper, we make the following contributions: (1) We develop a novel *static analysis* that characterizes the global memory behavior of a complete stream application. The static analysis can precisely identify the location and temporal reuse opportunities in most applications. The analysis is imprecise for a fraction of components whose control logic depends on the input data. In these cases, we provide a conservative but safe approximation of the application's memory behavior. (2) We propose a novel *layout algorithm* that leverages the identified location and temporal reuse opportunities along with the application's structure to optimize the memory layout. We incorporate code generation techniques that transform a stream program into efficient C code that effectively uses the generated memory layouts. (3) The memory optimizations are implemented as a new compiler for the StreamIt language. We evaluate the memory optimization on both Intel and ARM platforms using 14 benchmarks including three realistic mobile sensing applications. We compare our compiler against the standard StreamIt compiler with and without the cache optimization enabled [10]. We find that our optimizations reduce the memory footprint up to 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled.

2. STREAMIT OVERVIEW

Language Overview. Our work builds on the StreamIt programming language and compiler infrastructure (see [9] for a detailed description). The basic computation unit of StreamIt is a `filter` that may interact with other components by consuming data from the input channel, performing computations that may affect the state of the `filter`, and producing data on the output channel. StreamIt defines three basic memory operations: `pop`, `peek`, and `push`. The `peek` reads a sample at a given index in the input channel without consuming it, `pop` consumes a sample from the input, and `push` appends an item to the output channel. The `pop` and `push` access a channel sequentially while `peek` provides (limited) random access. Consistent with the SDF model, the number of items `peek`-ed, `pop`-ed, or `push`-ed during an execution is fixed and known at compile time. A filter has a `work` function that is executed each time the component is invoked. The `work` function specifies the rates r_{peek}, r_{pop}, and r_{push} for the `peek`, `pop`, and `push` instructions. A component may include states that are initialized using an `init` function.

StreamIt programs are written by hierarchically composing `filters` using `pipeline`, `split-join`, and `feedback` constructs into a Stream Flow Graph (SFG). The `pipeline` construct composes `filters` in sequence by connecting their inputs and outputs. The `split-join` construct distributes a stream to a parallel set of streams that are joined later. A `split` has a single input channel but multiple output channels. The `split` may either *duplicate* its input so that each parallel stream works on the same data or distribute it in a *round-robin* fashion. The `join` performs the opposite operation by taking data from multiple input channels and merging them into a single output channel in a *round-robin* fashion. The programmer may specify the number of elements to be produced by splitters and consumed by joiners during each invocation by providing a set of weights. The `feedback` construct is used for specifying feedback loops.

Execution Model. The problem of scheduling SDF components has been widely explored [11, 12, 13]. It is well understood that scheduling can have a significant impact on the program performance and memory utilization. A key advantage of SDF models is that they may be executed according to *cyclo-static* schedules. A cyclo-static schedule includes an initialization phase that is executed once and a steady phase that is executed repeatedly forever. The scheduling ensures that a `filter` is executed only if there is enough data on its input channel. In this paper, we assume a fixed schedule and do not consider the interaction between scheduling and memory optimization, which we will investigate in future work. We adopt the single appearance schedules (SASs) in [12], where a filter's work function appears only once in the schedule to reduce the code size.

The StreamIt language is tailored for stream programming. StreamIt follows SDF and adopts copy-by-value semantics and does not support pointers. These restrictions simplify the static analysis of stream programs and facilitate reasoning about their memory behavior statically.

3. DESIGN

ESMS builds on the properties of stream programs to optimize memory management. A key property of SDF systems is that their components may be executed using a cyclo-static schedule. Accordingly, the complete memory behavior of a program can be observed during an execution of the initialization phase followed by a *single* execution of the steady phase. This property is essential for providing a complete description of the memory behavior of the program. Furthermore, we propose a novel static analysis that leverages the semantics of stream programs to identify location and temporal sharing opportunities.

The layout algorithm uses the location and temporal sharing opportunities to allocate memory efficiently. The layout algorithm is based on two empirical insights regarding the memory operations of stream programs: (1) data sharing is often captured explicitly and may be exploited to reduce the memory footprint and number of memory accesses and (2) owing to the data flow structure, a filter may typically reuse the memory freed by its predecessor in the SFG. These insights coupled with three heuristics for handling memory conflicts form the basis of our layout algorithm. Code generation techniques are then employed to efficiently implement the derived layouts.

The subsequent sections detail the static analysis and layout algorithm. Empirical evidence regarding the effectiveness of our techniques is included in Section 4.

3.1 Static Analysis

The goal of our static analysis is to provide a sound approximation of the memory operations of a stream program under all possible executions. The memory behavior of the program is summarized as a hierarchical *Memory Graph* (MG)[1]. The top level of the graph is represented by components. Each component has one or more ordered input and output elements that form a *fragment*. Each *element* represents a sample that may be consumed or produced during the component execution. The grouped component inputs and outputs form the middle hierarchy while the elements constitute the bottom.

[1]An example memory graph is shown in Figure 2.

We distinguish two types of data reuse opportunities: *location sharing* and *temporal sharing*. Location sharing is captured by adding edges between elements to indicate when samples are passed without modification either within a component or between components. A property of MG is that the elements on any path can be stored at the same memory location, exposing location sharing. Temporal sharing is modeled separately by associating a live range L with each element that captures the time interval when an element is "live" given the fixed schedule. Obviously, elements that have non-overlapping live ranges may be stored at the same memory location.

The construction of MG proceeds in two steps:

- *Component Analysis:* Abstract interpretation (AI) is used to analyze the code of each component. The analysis constructs a fragment that captures the location and temporal sharing of a single component during an invocation of its `work` function.

- *Whole-program Simulation:* The schedule is simulated to stitch the previously constructed fragments and compose a MG that characterizes the entire application. The stitching process involves scaling component fragments to account for multiple invocations during the schedule, adjusting the live ranges of elements, and adding edges to capture data sharing between components.

The separation of the static analysis into two parts is motivated by the need to minimize the compile time. AI is significantly more expensive than the stitching process. Limiting the number of invocations of AI to one per filter allows us to handle multiple invocations of components.

3.1.1 Component Analysis

In this subsection, we present a static analysis that identifies location and temporal sharing during the component execution. Our analysis builds on AI initially developed by Cousot et al. [14]. The abstract domain of program variables is the intervals which approximate their concrete values. The functions α and γ map concrete values to intervals and vice-versa. The symbols $\perp$ and $\top$ represent the bottom (i.e., the empty set) and top (i.e., $[-\infty, +\infty]$ interval) of the real interval lattice. Aside from arithmetic operations, interval approximations may be defined over a broad range of functions [15] including those supported by StreamIt. Additionally, we use operators $\sqcup$ and $\sqcap$ to represent union and intersection of intervals respectively.

The `work` function of the component is represented as a control flow graph (CFG). The CFG has distinguishable entry and exit nodes, junction nodes with exactly two predecessors, branch nodes with a true successor and an optional false successor, and block nodes with one successor and predecessor. The junction nodes may be either simple or loop junctions. A block may contain multiple instructions, but we constrain each block to include a single `peek`, `pop`, or `push`. We will use I_i and O_k to denote the i^{th} input and k^{th} output elements of the considered component.

The state of a component may include local variables, global variables, and constant parameters known at compile time. The value of a global variable is maintained across component invocations. Since the derived results must hold for any component execution, states and the values of input elements are initialized to $\top$. In contrast, local variables are set to $\perp$ prior to their first assignment. Before analyzing the `work` function, constants including the component parameters are propagated.

The pseudocode for the analysis is included in Algorithm 1. The line numbers included in this section refer to this algorithm. The analysis extends the basic worklist algorithm, which updates a mapping from CFG edges to data flow facts until no new facts are derived. The notation $IN_v[n]$ and $OUT_v[n]$ refers to the data flow facts available immediately before and after a node n regarding variable v. Variables are interpreted over abstract intervals similar to the approach in [16]. However, our approach differs in two important aspects. First, to reduce pessimism, loop junctions are handled by unrolling loops rather than applying interval widening. We ensure termination by imposing an upper bound on the number of unrollings and applying widening when this bound is exceeded. The upper bound is set to the number of iterations if loop bounds are available or a large constant otherwise. Second, we handle function calls in a context-sensitive manner. The remainder of the discussion focuses on the unique aspects necessary for analyzing memory operations.

Temporal sharing. The analysis creates a fragment that includes r_{peek} input and r_{push} output elements. We determine the live range of each element to identify temporal sharing opportunities. At a high level, this requires determining when and which elements are referenced by a `peek`, `pop`, and `push`.

To keep track of the order of memory accesses, we add memory counters (mc) to the propagated data flow facts. The mc provides a time frame that captures the time when a memory operation is performed relative to the beginning of the component's execution. The mc of the entry node is initialized to zero. A memory operation increments the mc of the previous block (lines 11, 17, and 22). MCs are combined using the maximum at junctions (line 36).

To determine which elements are referenced by the memory instructions we define two sets – *pop* and *push* – that include the elements referenced by `pop`, `peek`, and `push`. The domain of the two counters is the concrete intervals. The *pop* and *push* are initialized to zero at the entry node and incremented after each block node that includes a `pop` (line 12) and `push` (line 23), respectively. The values of the *pop* and *push* counters are merged using *interval union* (lines 37 – 38) to track all possible referenced elements over all execution paths.

The derived live range facts of an element e are summarized in an interval $L[e]$. $L[e]$ is a global variable maintained during the analysis. We initialize $L[I_i] = [0, 0]$ since an input element I_i is live in the beginning of the component execution. In contrast, we set $L[O_k] = \emptyset$, since an output element O_k becomes live when it is first referenced by a `push`. For `pop`, `peek`, and `push`, the set *elem* contains the elements that are accessed by each instruction (lines 13, 18, and 24). The concretization function γ is used in the computation of *elem* to map an abstract interval representing these access indexes (i.e., IN_{pop} and OUT_{push}) to concrete values. The live range $L[e]$ grows monotonically by extending its interval to the current mc (lines 14, 19, and 25).

Location sharing. The analysis can also identify location sharing opportunities. Location sharing happens when a component reads an element I_i and passes it unmodified as

Input: CFG cfg
Output: Fragment f(V, E, L)
Data: worklist : queue
L – live ranges
A – pass or update
Γ – elements referenced in pop or push
P – potential edges

```
1  worklist = {cfg.entry()}
2  OUT_{mc,pop,push}[cfg.entry()] = 0
3  foreach e ∈ input do L[e] = [0, 0]
4  foreach e ∈ output do L[e] = ∅; A[e] = true;
5  P =∅
6  while worklist ≠ ∅ do
7    remove n from worklist
8    OUT_v[n] = IN_v[n] ∀v
9    switch n do
10     case [x = pop() | pop()]
11       if n is x = pop() then OUT_mc[n] = IN_mc[n] + 1
12       OUT_pop[n] = IN_pop[n] + 1
         // Live range computation
13       Let elem={ input[e] | e ∈ γ(IN_pop[n]) }
14       L[e] = L[e] ⊔ [IN_mc[n], IN_mc[n]]   ∀ e ∈ elem
         // Track location sharing
15       if n is x = pop() then OUT_Γ[x] = elem
16     case x = peek(y)
17       OUT_mc[n] = IN_mc[n] + 1
         // Live range computation
18       Let elem={ input[e] | e ∈ γ(IN_pop[n] + y) }
19       L[e] = L[e] ⊔ [IN_mc[n], IN_mc[n]]   ∀ e ∈ elem
         // Track location sharing
20       OUT_Γ[x] = elem
21     case push (x)
22       OUT_mc[n] = IN_mc[n] + 1
23       OUT_push[n] = IN_push[n] + 1
         // Live range computation
24       Let elem={ output[e] | e ∈ γ(IN_push[n]) }
25       L[e] = L[e] ⊔ [IN_mc[n], IN_mc[n]]   ∀ e ∈ elem
         // Track location sharing
26       A[e] = A[e] ∧(IN_Γ[x][n] ≠ ∅)   ∀e ∈ elem
27       P = P ∪{(e, f) | e ∈ IN_Γ[x][n] ∧ f ∈ elem }
28     case z = x ⊕ y
29       OUT_Γ[z][n] = ∅
30     case x = y
31       OUT_Γ[x][n] = OUT_Γ[y][n]
32     case branch
33       OUT^T_v[n] = IN_v[n] ∀v, if condition is true/undetermined
34       OUT^F_v[n] = IN_v[n] ∀v, if condition is false/undetermined
35     case simple junction
36       OUT_mc = max_{p∈pred(n)} OUT_mc[p]
37       OUT_pop[n] = ⊔_{p∈pred(n)} OUT_pop[p]
38       OUT_push[n] = ⊔_{p∈pred(n)} OUT_push[p]
39       OUT_Γ[x][n] = ⊔_{p∈pred(n)} OUT_Γ[x][p]   ∀ variables x
40     case loop junction
41       unroll the loop using widening if necessary
42   add the descendants whose facts have changed to the worklist
```

Algorithm 1: Component analysis

an output element O_k. Two constraints must be satisfied for location sharing: ($L1$) the element I_i must be passed as O_k in all executions, ($L2$) no other element I_j ($i \neq j$) is passed to O_k in any execution. Next, we will discuss how to determine whether these conditions hold.

Determining location sharing opportunities requires tracking how elements are passed from the input to the output. Let x be a variable whose value is set as a result of a pop or peek. Besides propagating facts regarding the value of x, our analysis also propagates the input element referenced by the pop or peek as $\Gamma[x]$. The values of $\Gamma[x]$ are propagated in assignments as long as x is not modified. Assignments such as y = x are handled by setting $\Gamma[y] = \Gamma[x]$ (line 31).

The analysis determines if the two location sharing constraints are satisfied during the interpretation of push (x).

	pop	*push*	*mc*
$a_{1,2}$	0	0	0
n_2	$L[I_0] = [0,0] \sqcup [0,0] = [0,0]$ $x = \top$ $\Gamma[x] = \{I_0\}$		
$a_{2,3}$	1	0	1
$a_{3,4}$	1	0	1
n_4			
$a_{4,5}$	2	0	1
n_5	$L[O_0] = \emptyset \sqcup [1,1] = [1,1]$ $A[O_0] = \texttt{false}$		
$a_{5,8}$	2	1	2
$a_{3,6}$	1	0	1
n_6	$L[I_1] = [0,1]$ $t_1 = \top$ $\Gamma[t_1] = \{I_1\}$		
$a_{6,7}$	2	0	2
n_7	$L[O_0] = \emptyset \sqcup [2,2] = [2,2]$ $A[O_0] = \texttt{true}$ $P = \{I_1, O_0\}$		
$a_{7,8}$	2	1	3
n_8	$L[I_0] = [0,0]$ $L[I_1] = [0,1]$ $L[O_0] = [1,1] \sqcup [2,2] = [1,2]$ $A[O_0]$=false $P = \{I_1, O_0\}$		
$a_{8,9}$	2	1	3
n_9	$L[I_2] = [0,0] \sqcup [3,3] = [0,3]$ $t_2 = \top$ $\Gamma[t_2] = \{I_2\}$		
$a_{9,10}$	3	1	4
n_{10}	$L[O_1] = \emptyset \sqcup [4,4] = [4,4]$ $A[O_1]$=true $P = \{(I_1, O_0), (I_2, O_1)\}$		
$a_{10,11}$	3	2	5

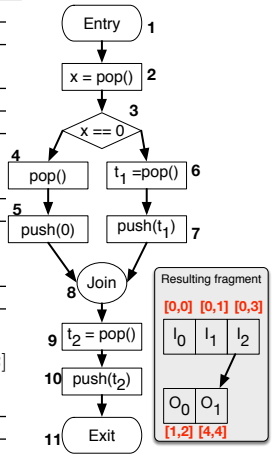


Figure 1: Abstract interpretation of a code fragment

The analysis determines whether the argument x is passed ($\Gamma[x] \neq \emptyset$) or updated ($\Gamma[x] = \emptyset$) by inspecting $\Gamma[x]$. The analysis ensures that this property holds across all paths by associating a variable $A[O_k]$ with each output element O_k. $A[O_k]$ is initialized to *true* and set to *false* if there exists an execution where O_k is updated rather than passed. Condition ($L1$) is satisfied for element O_k when $A[O_k]$ is *true*. Additionally, we keep track of all potential passes from input element I_i to output element O_k during all executions in variable P. Condition ($L2$) is satisfied for an pair (I_i, O_k) if $(I_i, O_k) \in P$ is the only edge. The edges that satisfy both conditions are added as the final edges (E) of the fragment.

An example of AI on a CFG fragment is shown in Figure 1. The table shows the data flow facts available on each arc before and after the interpretation of a node. Only the updated or newly derived facts are shown in the table. Initially, the values of the *mc*, *pop*, and *push* are set to zero. Interpreting the pop on node 2 updates the fact that I_0 is live in the interval $L[I_0] = [0,0]$. Since the value of I_0 is unknown during the analysis, x is set to $\top$ and $\Gamma[x]$ is set to I_0. The truth-value of the condition at branch 3 cannot be determined, so the analysis will execute both branches. The pop in node 4 simply indicates that I_1 is not in use anymore and, as a result, no new facts are derived. The interpretation of the push in node 5 results in increasing the *push* and *mc* counters. Additionally, $L[O_0]$ that was initially $\emptyset$ is updated to $[1,1]$. The algorithm records that O_0 was updated rather than passed by setting $A[O_0] = false$. I_1 is saved in t_1 in node 6 and passed as O_0 in node 7. Thus during the execution of node 7, the analysis checks for location sharing. Since $t_1 = I_1$, the analysis sets $A[O_0] = true$ and adds (I_1, O_0) as a potential location share. The pop and push in nodes 6 and 7 are handled similarly. The analysis handles join nodes by using either the maximum, interval union, or "and" operator. There are two interesting cases. The live ranges $L[O_0]$

are merged to be $[1, 2]$ using the interval union. Similarly, $A[O_0]$ has different values indicating that O_0 was passed on one path but updated on the other. Thus, the combined value of $A[O_0]$ is set to $false$ using the "and" operator. The analysis continues producing the results shown in the figure.

3.1.2 Whole-program analysis

The component analysis constructs fragments that describe the memory optimization opportunities during a single invocation of a component. The whole-program analysis stitches these fragments to compose a MG that characterizes the memory operations of the entire program. We remind the reader that a stream schedule is composed of an initialization phase executed once and a steady phase that is executed repeatedly forever. To characterize the entire program it is sufficient to simulate the initialization phase and a single execution of the steady phase.

The stitching algorithm considers the execution of components in the schedule order. Consider the execution of a component that is invoked n times during the schedule. For each component, we add $|I| = r_{peek} + r_{pop} \times (n-1)$ input elements and $|O| = r_{push} \times n$ elements in the MG.

The stitching algorithm must relate the indexes of input/output elements in MG to those in the memory fragment (F). The mapping must recognize that a component may access its input over overlapping windows and produce samples in overlapping windows. Accordingly, the memory operations of I_i is the union of memory operations of input elements I_j^F such that $I_i \in I_{MG\to F}(I_i)$:

$$I_{MG\to F}(I_i) = \{I_j^F \geq 0 \mid I_j^F = I_i - k \times r_{pop} \quad k \in \mathbb{N}\}$$

Similarly, the memory operations of O_k are the same as the operations of O_l^F such that:

$$O_{MG\to F}(O_k) = \{O_l^F | O_k = \mod (O_l^F, r_{push})\}$$

Location sharing opportunities are computed by iteratively considering each possible link (I_i, O_k). A link (I_i, O_k) is added to MG if there exists a link (I_j^F, O_l^F) in the fragment such that $I_j^F \in I_{MG\to F}(I_i)$ and $O_l^F \in O_{MG\to F}(O_k)$. The construction does not introduce location sharing conflicts regardless of potentially overlapping output windows.

The next step in the construction of MG is to capture the exchange of data between components. This process takes advantage of the hierarchical nature of the stream graph. The components connected in a pipeline are adjacent in topological order and linked through their inputs and outputs. The split-join constructs are handled by adding split and join components that internally implement either duplicate data or round-robin policies.

The live ranges included in the fragment are the time when each memory operation was performed relative to the beginning of the component execution. A live range in MG is represented as a triple $(phase, step, mc)$ where the $phase$ is the phase of the schedule (0 for init, 1 for steady), $step$ is a counter that is incremented after each component invocation, and mc is the memory counter in the fragment. The interval union operator can be easily extended to operate over triples. To account for window overlaps, the live ranges of an element I_i is set to equal $\bigsqcup_{I_j^F \in I_{MG\to F}(I_i)} L[I_j^F]$. Additionally, we also need to account that some elements may be shared. An element e shares the same location with f if there is a path between them. The live range of e is expanded to $\bigsqcup_{f \in shared_{MG}(e)} L[f]$ to account for location sharing.

3.2 Memory Layout

The layout algorithm operates on a single-appearance schedule [12]. Generating a good layout is necessary for reducing memory usage, improving performance by reducing memory accesses, ensuring good cache locality, and generating efficient code. We propose three heuristics and our empirical evaluation shows they effectively balance these requirements. Our approach is driven by two empirical insights into stream programs. (1) StreamIt and other data flow languages include constructs such as split-joins that share and reorder samples. Traditionally, memory copying across channels are used to implement these constructs. We opt for the alternative of changing the logical layout of samples without performing any copy operations. This leads to significant reductions in both the memory size and number of accesses. Samples are reordered using round-robin split-joins can often be accessed efficiently using linear iterators of the form $base + step \times i$. Typically, $step$ is a small constant leading to reasonable cache locality. (2) A filter operates on the input provided by the previous component in the SFG. It is often possible for a filter to reuse the memory allocated for the previous filter. This is because as a filter `pops` samples from the input, the memory locations where the samples were located become available for reuse.

Prior work has considered different buffer management strategies for storing samples due to sliding windows. StreamIt filters process their input in sliding windows by peeking more samples than r_{pop}. The proposed techniques include modulation [10, 17] and copy-shift [10]. The modulation strategy stores samples in a circular buffer and requires modulo operations for indexing. Copy-shift avoids potentially expensive modulo operations by shifting the remaining samples of sliding windows in the buffer from the previous filter execution. When coupled with execution scaling, the copy-shift approach can significantly outperform modulation [10]. Execution scaling works by scaling the number of times each component in a schedule is executed. This has the effect of reducing the number of copy operations relative to the size of the input. Our memory management approach uses the copy-shift buffer management.

We organize the memory M as cells, each cell having an address and a size equal to the machine word (64 bits on considered platforms). The memory supports two operations: `memappend` and `meminsert`. The `memappend(`w`)` operation allocates w words at the end of current memory allocation. The `meminsert(`$addr$`,` w`)` operation inserts w words at location $addr$, reindexing the memory addresses to account for the insertion. Additionally, `meminsert` ensures that the mapping between elements and cells is maintained after the insertion. A layout is defined as a mapping $\Pi : V \to M$ that maps each element in MG to a memory location. We also maintain the reverse mapping $\Phi : M \to 2^{|V|}$ from an address to a set of elements stored at the address. A valid mapping ensures that for any memory location m, the intersection of the live ranges of the elements in $\Phi(m)$ is empty. This is accomplished by inspecting the appropriate live ranges in MG.

The layout algorithm generates the memory layout by simulating the execution of a component in scheduling order and constructing Π incrementally. The pseudocode is included in Algorithm 2 and the line numbers included in this section refer to this algorithm. The input to the algorithm is the MG and the SFG with appropriately defined successor

```
Input: MG(V, E, L) – memory graph
       SFG - stream flow graph
       R[c] – elements that are remainders of component c
       store[c] – storage for the remainders of component c
Data: in[prev] – input windows from the prev component(s)
      out[next] – output windows to the next component(s)
1  foreach (phase, n, c) in schedule.init do
2    if c is source then
3      out = newwindow(r_push[c] × n )
4      p = memappend (r_push[c] × n)
5      foreach i = 0 . . . n do Π[out(i)] = p(i)
6    else
7      switch c do
8        case duplicate-split
9          in = flatten (pred(c).out[c])
10         foreach next ∈ succ(c) do
11           out[next].append(in)
12       case round-robin-split
13         in = flatten (pred(c).out[c])
14         S = vector of weights of the splitter
15         start = 0
16         for 0 . . . n do
17           foreach next ∈ succ(c) do
18             out[next].append(in[start:start+S[next])
19             start = start + S[next]
20       case round-robin-join
21         J = vector of weights of the joiner
22         next = succ (c)
23         foreach prev ∈ pred(c) do start[prev] = 0
24         for 0 . . . n do
25           foreach prev ∈ pred(c) do
26             w = prev.out[c])
27             out[next].append(w[start[prev]:start[prev]+J[prev]])
28             start[prev] = start[prev] + J[prev]
29       case filter
30         windows = window(in, r_peek[c], r_pop[c])
31         for e ∈ R[c] do
32           Φ[e] = store[c][e]
33         next = succ(c)
34         out = newwindow(r_push × n)
35         j = 0 // index of the output sample;
36         for w = 0. . . n do
37           U = sharedElements(windows[w])
38           for i ∈ U \ R[c] do
39             shared = Φ[Π[i]]
40             hasConflict = ⋂_{e∈shared} L(e)
41             if hasConflict = ∅ then
42               Π[j] = Π[i];
43               j++;
44             else
45               ... handle conflicts using described heuristics ...
46   Procedure sharedElements(window)
47     U = {I_i | I_i ∈ window}
48     for I_i in window do
49       L = {I_i|(I_i, O_k) ∈ E}
50       for e_o in L do
51         Π[e_o] = Π[I_i]
52         U = U \{e_o}
53     return U
```

Algorithm 2: Layout algorithm

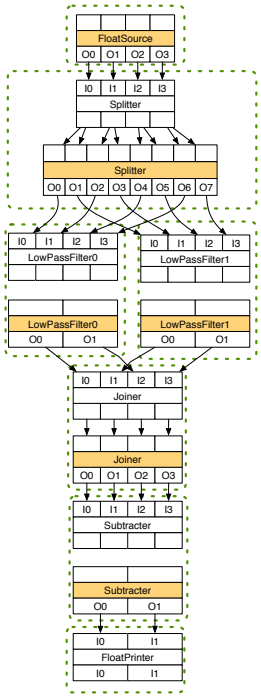
and predecessor functions. Components operate over windows of input or output elements. We define two operations for manipulating windows: `flatten` and `window`. The `window`(w, *size*, *overlap*) transforms a single window into a group of windows that overlap by *overlap* elements with each window having *size* elements. Conversely, `flatten` produces a single window from overlapped windows. If c is a source, the algorithm creates a window of size $r_{push} \times n$ that is mapped to the next available memory location (line 2). All other components will generate their layouts based on the windows of the previous components in the SFG, the details depending on whether the component is a `split`, `join`, or a `filter`.

Split-joins are used to share and to reorder samples and their behavior is captured as edges in MG. The first step in handling both splits and joins is to flatten the output window(s) from the previous component(s) into a single window. If c is a duplicate splitter, the layout algorithm will pass the input window to the each of its successors (lines 10 – 11). If c is a round-robin splitter, the layout algorithm passes a subset of the elements in the input window to the output window of each successor (lines 14 – 19). The number of elements passed to each successor is part of the splitter specification in the program (stored in S). Round-robin joiners are handled in an equivalent manner. The layout generates an output window in which elements from each one of the predecessor components are appended (lines 21 – 28). Samples are inserted by considering the predecessors in order and adding the programmer specified number of samples (stored in J). Note that `split` and `join` operate in the logical space and do not require changing the mapping between logical space and memory.

The layout process for a filter starts by flattening the output window of the previous component and windowing the result according to the component's peek and pop rates. This generates n windows each containing r_{peek} samples that the algorithm will manipulate. The algorithm will first consider each element I_i in the input window, determining if there is an edge (I_i, O_k) in MG, where O_k is an output element. The existence of the edge indicates that I_i and O_k may share the same memory location. Accordingly, we map O_k to I_i's memory cell (i.e., $\Pi[O_k] = \Pi[I_i]$). These operations are performed as part of `sharedElements` procedure in the pseudocode. Let U include the set of elements whose mapping has not been determined. The elements in U are updated by the component during its execution. We consider three heuristics for laying out these conflicts: *always-append*(AA), *append-on-conflict*(AoC), and *insert-in-place*(IP). (1) The AA heuristic maps elements in U to a group of contiguous memory cells at the end of the current allocation. This approach has the advantage of ensuring cache locality and simplifies the generated code. (2) The AoC heuristic will first try to layout windows within the memory region allocated for the previous component. When this is not possible due to conflicts in the live ranges of variables, then the window will be mapped to a contiguous portion of memory at the end of current allocation. We expect that this heuristic will reduce the size of memory allocation over the previous heuristic albeit at the cost of increased code complexity and execution time. (3) The IP heuristic inserts a number of memory cells at the location where a conflict is determined. This has the effect of allowing the subsequent components to operate on a layout that maps their input elements to proximate memory locations.

The layout algorithm must account for the fact that after the initialization phase of the schedule and at the completion of each steady phase, components with $r_{peek} > r_{pop}$ will have input elements that are used in subsequent executions. The remainders of a component c are stored in $R[c]$. Consistent with the copy-shift strategy, such components are responsible for saving these remainders at the completion of the initialization and steady phases. The re-

mainders are loaded in the beginning of the input window of a component prior to the beginning its execution. Remainders are treated as a special case since their live ranges cover the entire phase, creating few opportunities for temporal reuse. Accordingly, remainders are saved and loaded from special remainder stores. Components that operate on shared buffers also have shared remainders. We optimize the loading and storing of data from shared stores to avoid duplicate memory operations.



Schedule
Init: (FloatSource,3) (Splitter,3) (LowPassFilter0,1) (LowPassFilter1,1) (Joiner,1) (Subtractor,1) (FloatPrinter,1)
Steady: (FloatSource,1) (Splitter,1) (LowPassFilter0,1) (LowPassFilter1,1) (Joiner,1) (Subtractor,1) (FloatPrinter,1)

Live ranges:

Element	Start	End
FloatSource:O_0	(0, 0, 0)	(0, 7, 3)
FloatSource:O_1	(0, 1, 0)	(1, 3, 3)
FloatSource:O_2	(0, 2, 0)	(1, 6, 1)
FloatSource:O_3	(1, 0, 0)	(1, 6, 2)
LowPassFilter0:O_0	(0, 6, 4)	(0, 9, 3)
LowPassFilter0:O_1	(1, 2, 4)	(1, 5, 3)
LowPassFilter1:O_0	(0, 7, 4)	(0, 9, 4)
LowPassFilter1:O_1	(1, 3, 4)	(1, 5, 4)
Subtractor:O_0	(0, 9, 2)	(0, 10, 0)
Subtractor:O_1	(1, 5, 2)	(1, 6, 0)

Physical layout:

MEM	Initialization	Steady
0	FloatSource:O_0 LowPass1:O_0	FloatSource:O_1 LowPass1:O_1
1	FloatSource:O_1	FloatSource:O_2
2	FloatSource:O_2	FloatSource:O_3
3	LowPass0:O_0	LowPass0:O_1
4	Subtractor:O_0	Subtractor:O_1

Remainder store: 2 cells

Figure 2: Bandpass filter: memory graph, stream schedule, and physical layout

Figure 2 shows the memory graph generated and stream schedule for a bandpass filter. The figure also includes the physical layout generated using the AoC heuristic. In the initialization phase, the algorithm starts by creating a window containing 3 elements that start at MEM[0]. The duplicate splitter replicates this window to the input of the low pass filters. `LowPassFilter0` checks if it may reuse the cell allocated for `LowPassFilter0:`I_0 to store its `LowPassFilter0:`O_0. Since the intersection of the live ranges $[(0,0,0),(0,7,3)]$ and $[(0,6,3),(0,9,3)]$ is not empty, this is not possible. This is expected since this sample is also an input to `LowPassFilter1`. Accordingly, it is mapped to MEM[3]. `LowPassFilter1` performs a similar check to determine if the cell allocated for `LowPassFilter1:`I_0 may be reused for `LowPassFilter1:`O_0. In this case where the intersection of the live ranges $[(0,0,0),(0,7,3)]$ and $[(0,7,4),(0,9,4)]$ is empty, the cell may be reused. Accordingly, `LowPassFilter1:`O_0 will reuse MEM[0]. In the steady phase, `FloatSource:`O_3 is pushed and shifted to MEM[2] because `FloatSource:`O_1 and `FloatSource:`O_2 will be loaded to MEM[0:1]. The following filters produce samples at the locations as in the initialization phase, proving the mapping allows to execute the steady schedule infinite often. The resulting layout is shown in Figure 2. We note that a naive memory management approach that uses respective buffers for splits and joins (as it is the case for the default StreamIt compiler) would require a total of 12 cells, one for each output element. In contrast, our layout algorithm requires only (5+2) cells by taking advantage of location and temporal sharing.

Benchmark	Description
AutoCor	autocorrelation computation
Bitonic Sort	bitonic sort
MergeSort	merge sort
Repeater	repeats odd samples M times
FIR/FIRCourse	finite impose response filters
FMRadio	FM radio with 10-way equalizer
FFT2/FFT3	FFT computation
MatrixMult/MatrixMultBlock	matrix multiplication
BeepBeep [18]	acoustic localization
MFCC	computation of MFCC coefficients
Crowd [1]	estimation of number of co-located speakers

Table 1: Benchmark suite

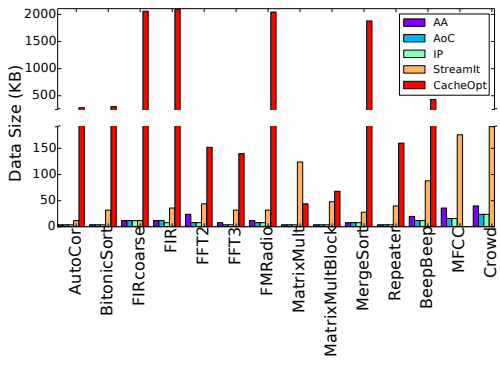
Code generation. The last step in ESMS is the generation of C code from a StreamIt program. Most of the details of code generation are unsurprising. The only aspect that requires careful handling is the generation of code for `peek`, `pop`, and `push`. When the memory is accessed contiguously, generating code for memory instructions is straightforward. However, handling a fragmented memory layout is challenging when memory operations are nested in loops. Our compiler implements two methods for handling this case. In most cases, we opt for splitting the loop at the boundary of contiguous memory locations. Obviously, this trade-off increased code complexity for execution time improvements. The alternative is to apply indirect addressing where the memory instructions operate in the logical space and a static mapping between the logical and physical space is concretized as a lookup table used at run-time. A tunable constant is used to control between the two options.

4. EXPERIMENTS

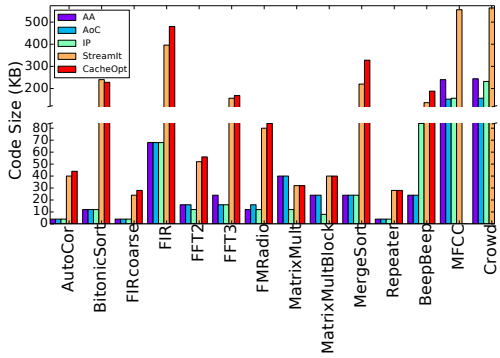
In the following, we show the benefits of the proposed static analysis and memory layout algorithm. Benchmarks are made on a desktop machine that has a 3GHz Intel(R) Xeon(R) CPU E5-1680 v2, and a Nexus 10 tablet that has a Samsung Exynos 5250 SoC with 1.7 GHz Dual-core Cortex-A15. The Xeon has 32KB L1 instruction and data caches, 256K L2, and shared 25MB L3 caches. The ARM has 32KB L1 instruction and data caches, and a shared 1 MB L2 cache. The system runs Android 5.1. Programs are compiled using the native development kit (NDK) r10d that uses `gcc 4.8`. A wrapper Java application is generated to invoke the generated code. All programs are compiled using the highest optimization level (i.e., `-O3`).

The suite of benchmarks consists of 14 stream applications (see Table 1). The majority of the benchmarks were developed as part of the StreamIt project. In addition, we implemented three mobile sensing applications using StreamIt: the `BeepBeep` app [18] performs sound-based localization, the `MFCC` app implements the core of a speaker identification app (e.g, [2]), and the `Crowd` app [1] estimates the number of co-located speakers.

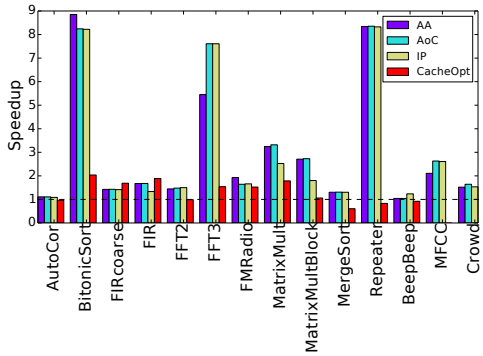
Static Analysis. We have evaluated the pessimism of the static analysis proposed in Section 3.1. The static analysis was able to precisely characterize the memory behaviors of all benchmarks with the exception of the MergeSort benchmark. It includes nondeterminism since the control flow depends on the input data. This shows that for a wide range of programs our analysis can precisely characterize the complete memory behavior of a stream program. For the



(a) Data size



(b) Code size



(c) Speedup

Figure 3: Data, code, and speedup improvements on Intel Core i5-3550

MergeSort benchmark, the analysis derived a safe approximation of location and temporal sharing opportunities. We note that even for these benchmarks, the nondeterminism is confined to a single component and does not affect the others in the application.

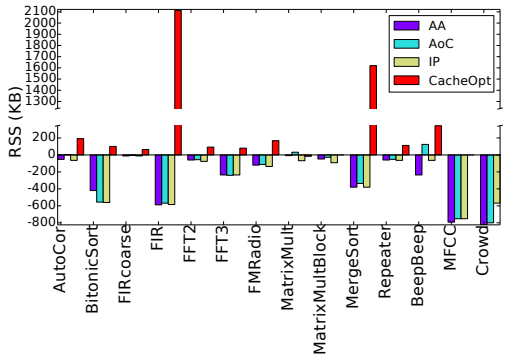
Intel Measurements. We have measured the impact of memory optimizations on three dimensions: data size, code size, and speedup. We report both absolute and relative improvements in these dimensions. The relative improvements are computed based on the performance of the default StreamIt compiler (labeled `StreamIt`). We have also run the StreamIt in conjunction with the cache optimization described in [10]. These results are reported as `CacheOpt`. The compiler with cache optimizations failed to generate code for `MFCC` and `Crowd` applications because it ran out of memory. The performance of the *always-append*, *append-on-conflict*, and *insert-in-place* heuristics are denoted by labels `AA`, `AoC`, and `IP` respectively. The code and data size results were obtained only on the Intel perform using the `size` utility. The tool reports both the code and data sizes in multiples of a page. The speedup results are based on the CPU user time reported by the `time` utility.

Figure 3a shows the data size with `StreamIt` as the baseline. The average reductions on memory footprint of `AA`, `AoC`, and `IP` are 50KB, 55KB, and 98KB. These represent reductions on data size between 45–96%. The `AA` heuristic provides the smallest reductions on data size since it always appends rather than attempts to resolve memory conflicts. The `AoC` and `IP` heuristics achieved comparable performance in terms of memory usage. In contrast, enabling the `CacheOpt` increased memory consumption by an average of 627KB. This increase can be as large as 98% for MergeSort or FFT2.

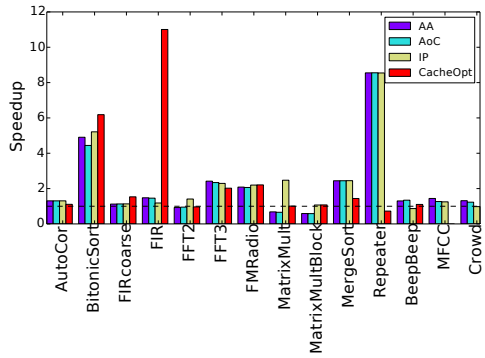
Figure 3b shows the code size with `StreamIt` as the baseline. The average code reductions for `AA`, `AoC`, and `IP` are 130KB, 143KB, 136KB. In relative terms, the average reductions are 69%, 72%, and 77% respectively. The ESMS optimizations reduce the code size by not generating code for `split-join` constructs and other components that reorder (without modifying) the input data. Nevertheless, even with these savings, there are cases when ESMS has larger code size than `StreamIt`. `CacheOpt` typically has a minimal impact on the code size. On average, it adds 14KB to the code size of the considered benchmarks.

Figure 3c shows the speedup relative to `StreamIt`. The average speedups of `AA`, `AoC`, and `IP` are 3, 3.1, and 3. In contrast the average speedup of `CacheOpt` is merely 1.07. All of ESMS heuristics outperformed `CacheOpt` with the exception of the two FIR benchmarks on both platforms because the FIR pipeline is long enough to cause instruction cache misses in one schedule iteration and the CacheOpt performs better by reducing the cache miss rate with execution scaling which is constrained by the data cache size. ESMS with reduced data size is expected to support more execution scaling in the future. Otherwise, the reason for these significant performance improvements is the fact that ESMS uses less memory access by effectively sharing data across components. We validated that this was the case by using `cachegrind` to track the number of memory accesses (data not included due to limited space). Moreover, the smaller footprint leads to a smaller working set that fits within the large cache of this platform. On the Intel platform, ESMS managed to improve the stream processing throughput while significantly reducing the memory consumption. The heuristics that handle conflicts either by inserting or appending achieve more memory savings than `AA`. Resolving conflicts through insertion tends to achieve more reductions on data size but slightly lower performance than appending.

ARM Measurements. We generate Android applications that include the compiled code as a shared library. To quantify the impact of memory optimizations on Android applications, we measure the maximum resident working set size (RSS), which includes the total memory allocation at run-time for both the Android application and the shared library. Figure 4a shows the RSS relative to the baseline StreamIt on ARM. The average reductions for `AA`, `AoC`, and `IP` are 274KB, 247KB, and 261KB respectively. In contrast,



(a) RSS relative to baseline



(b) Speedup

Figure 4: RSS and speedup improvements for Samsung Exynos 5250

`CacheOpt` increased the buffer size by 347KB in order to operate on larger buffers. This shows ESMS facilitates effective memory savings. Figure 4b shows the speedup on ARM. The average speedups for `AA`, `AoC`, and `IP` are 2.18, 2.03, and 2.3, respectively. `CacheOpt` achieved a comparable speedup of 2.18. The standard deviation for the speedup improvements for `AA`, `AoC`, `IP`, and `CacheOpt` were 2.12, 2.16, 2.3, and 2.95. The lower standard deviation of `AA` indicates that it performed the best. ESMS achieves similar performance but significantly reduces the memory footprints.

5. RELATED WORK

The memory optimization problem is similar to the classical aggregate update [19] problem in functional programming languages. The numerous solutions proposed to address this problem can be broadly classified as either run-time or static approaches. Run-time approaches typically rely on either garbage collection or reference counting. In contrast, static approaches require compiler analyses to determine live ranges of variables in order to ensure safe data sharing. Live range information may be extracted either through heuristics [20] or abstract interpretation [21]. A distinguishing aspect of our analysis is that it takes advantage of stream properties to characterize *complete* applications.

In [22], a greedy in-place reuse of memory allocations is proposed for the data flow model of LabView. The heuristic approach chooses variables using a cost metric to merge and store them in the same location. In contrast, our layout algorithm takes advantage of the structure of the data flow for memory optimization. More recently, an annotation-based approach has been proposed to address memory management in the context of data flow languages [23]. We note that StreamIt includes explicit data sharing information as part of the `split-join` construct. Moreover, we show that significant improvements in stream processing rates and reductions on memory footprints may be achieved without requiring annotations.

The problem of scheduling SDF graphs to optimize different metrics has been studied extensively [11, 17]. The previous work of phased scheduling [12] shows a steady state SDF schedule can be rearranged in phases to shorten the output latency compared with SASs which have only one phase at the expense of increasing code size. Since our goal is to optimize memory usage and performance, ESMS may support phased scheduling in the future.

On the other hand, several cache performance improvements were proposed including the copy-shift buffering and execution scaling for StreamIt in [10], and cache-aware optimizations for synchronous data flows [24] as part of the Ptolemy project [25]. In contrast with [10] that trades space for performance, our approach improves cache locality by saving space while eliminating unnecessary memory operations to improve the performance. In [17], the memory reuse is based on overlaying channel buffers in terms of their live ranges while maintaining periodical modulo access. Compiler optimizations were also considered to generate instructions to avoid the modulo overhead selectively given the static schedule. From this perspective, our compiler optimizations allow for more aggressively reuse and even remap non-contiguous memory accesses across filter invocations at some cost of increasing code complexity.

In addition, the linear analysis in [26] is an effective alternative to improve the performance by reducing the number of linear filters while saving memory usage accordingly. Instead, we consider filters only in terms of general memory operations and are able to eliminate non-productive filters such as split-joins or filters that reorder their input without modifications. In this sense, our optimizations are not limited to linear filters but more general.

Memory management can have a significant performance impact on stream processing engines. XStream [6] and WaveScript [8] use an abstract data structure called SigSeg to merge and segment data streams efficiently. CSense [5] proposes several memory management techniques to optimize the exchange of frames between components. StreamFlex [27] is yet another stream processing toolkit written in Java, aiming at avoiding the garbage collection overhead while satisfying real-time constraints.

6. CONCLUSIONS

In this paper, we presented – ESMS – a novel approach for optimizing the memory management of stream programs. Our approach leverages the unique properties of stream programs for both static analysis and memory layout. We developed a novel static analysis technique that characterizes the *behavior* of complete stream programs by identifying location and temporal sharing opportunities. Our analysis scales to handle large stream programs by separating the components analysis from the creation of memory graphs through stitching. An evaluation conducted on 14 benchmarks including three for mobile sensing applications reveals that the analysis is precise for a majority of stream applications.

Besides, sound approximations of the memory behavior are provided for the other applications.

We developed a novel memory layout algorithm. The algorithm recognizes that stream programs have significant opportunities for location sharing. In StreamIt, these opportunities are often the result of constructing programs using `pipeline` and `split-join` constructs. Additionally, we observe that connected filters in a SFG may often operate on the same memory since the live ranges of their buffers usually do not overlap. Obviously, this is not always possible. We introduced three simple heuristics to handle conflicts when they arise during the memory layout process. Our empirical evaluation indicates that ESMS may achieve significant memory savings. On the Intel platform, these memory savings are coupled with improvements in stream processing rates over StreamIt with cache optimizations. On the ARM platform, the stream processing improvements are comparable to those achieved by StreamIt with cache optimizations. These results show that ESMS is effective in developing efficient memory management for stream programs. In the future, we plan to continue exploring compiler optimization techniques to further improve the performance of stream processing engines for mobile sensing applications.

7. ACKNOWLEDGMENTS

This work supported by the National Science Foundation (Grant No. 1144664) and by the Roy J. Carver Charitable Trust (Grant No. 14-4355).

8. REFERENCES

[1] C. Xu, S. Li, G. Liu, Y. Zhang, E. Miluzzo, Y.-F. Chen, J. Li, and B. Firner, "Crowd++: Unsupervised speaker count with smartphones," in *Ubicomp*, 2013.

[2] E. Miluzzo, C. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. Campbell, "Darwin phones: the evolution of sensing and inference on mobile phones," *MobiSys*, 2010.

[3] M. Lin, N. D. Lane, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell *et al.*, "Bewell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization," in *Wireless Health*.

[4] N. Nikzad, N. Verma, C. Ziftci, E. Bales, N. Quick, P. Zappi, K. Patrick, S. Dasgupta, I. Krueger, T. v. Rosing, and W. G. Griswold, "Citisense: Improving geospatial environmental assessment of air quality using a wireless personal exposure monitoring system," in *Wireless Health*, 2012.

[5] F. Lai, S. S. Hasan, A. Laugesen, and O. Chipara, "Csense: A stream-processing toolkit for robust and high-rate mobile sensing applications," in *IPSN*, 2014.

[6] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "XStream: a Signal-Oriented Data Stream Management System," in *ICDE*, 2008.

[7] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, "The jigsaw continuous sensing engine for mobile phone applications," in *SenSys*, 2010.

[8] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," in *LCTES*, 2008.

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, 2002, pp. 179–196.

[10] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache aware optimization of stream programs," *ACM SIGPLAN Notices*, 2005.

[11] S. S. Bhattacharyya and E. A. Lee, "Scheduling synchronous dataflow graphs for efficient looping," *Journal of VLSI Signal Processing*, vol. 6, no. 3, pp. 271–288, Dec. 1993.

[12] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased scheduling of stream programs," in *LCTES*, 2003.

[13] E. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *Computers, IEEE Transactions on*, vol. C-36, no. 1, pp. 24–35, 1987.

[14] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, 1977.

[15] M. Daumas, D. Lester, and C. Muoz, "Verified real number calculations: A library for interval arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226–237, Feb 2009.

[16] P. Causot and R. Cousot, "Static Determination of Dynamic Properties of Programs," in *SIGSOFT*, vol. 2, no. 2, 1977, pp. 77–94.

[17] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *Signal Processing, IEEE Transactions on*, vol. 42, no. 5, pp. 1190–1201, 1994.

[18] C. Peng, G. Shen, Y. Zhang, Y. Li, and K. Tan, "Beepbeep: a high accuracy acoustic ranging system using cots mobile devices," in *SenSys*, 2007.

[19] P. Hudak and A. Bloss, "The aggregate update problem in functional programming systems," in *POPL*, 1985.

[20] P. Schnorf, M. Ganapathi, and J. L. Hennessy, "Compile-time copy elimination," *Software: Practice and Experience*, vol. 23, no. 11, pp. 1175–1200, 1993.

[21] M. Odersky, "How to make destructive updates less destructive," in *POPL*, 1991.

[22] S. Abu-Mahmeed, C. McCosh, Z. Budimlić, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup, "Scheduling Tasks to Maximize Usage of Aggregate Variables in Place," 2009.

[23] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages," *LCTES*, 2012.

[24] S. Kohli, "Cache aware scheduling for synchronous dataflow programs," UC Berkeley, Tech. Rep., 2004.

[25] E. A. Lee., "Overview of the ptolemy project," UC Berkeley, Tech. Rep., 2003.

[26] A. A. Lamb, W. Thies, and S. Amarasinghe, "Linear analysis and optimization of stream programs," in *PLDI*, 2003.

[27] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: High-throughput stream programming in java," *SIGPLAN Not.*, 2007.