Buffer Optimization and Dispatching Scheme for Embedded Systems with Behavioral Transparency [*]

Jiwon Hahn
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
jhahn@uci.edu

Pai H. Chou
University of California, Irvine, USA, and
National Tsing Hua University, Taiwan
phchou@uci.edu

ABSTRACT

Software components are modular and can enable post-deployment update, but their high overhead in runtime and memory is prohibitive for many embedded systems. This paper proposes to minimize such overhead by exploiting behavioral transparency in models of computation. In such a model (e.g., synchronous dataflow), the state of buffer requirements is determined completely by the firing sequence of the actors without requiring functional simulation of the actors. Instead of dedicating space to each channel or actor statically, our dispatcher passes buffer pointers to an actor upon firing. Straightforward implementations are counterproductive, as fine-grained allocation incurs high pointer overhead while coarse-grained allocation suffers from fragmentation. To address this problem, we propose medium-grained, "access-contiguous" buffer allocation scheme. We formulate the problem as 2-D tiles that represent the lifetime of the buffers over time and define operators for their translation and transformations to minimize their memory occupation spatially and temporally. Experimental results on real-life applications show up to 70% data memory reduction compared to existing techniques. Our technique retains code modularity for dynamic configuration and, more importantly, enables many more applications that otherwise would not fit if implemented using previous state-of-the-art techniques.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; B.3.2 [**Memory Stuctures**]: Performance analysis and design aids—*Simulation, Optimization*

General Terms

Algorithms, Design, Experimentation

Keywords

Behavioral transparency, runtime system, model of computation, component software, memory optimization, buffer allocation

[*]Research sponsored by NSF CAREER Award CNS-0448668.

1. INTRODUCTION

The ability to organize software as modular, dynamically loadable components is becoming critical. Once deployed, it is difficult and costly if not impossible to retrieve these systems from the deployment site to perform firmware update or other maintenance tasks individually. Also, it is desirable to support higher level programming and code reuse in terms of invocation of a stable API provided by the platform. This way, the user program can be kept small, and an update entails changing sequencing and parameter values, rather than re-implementation of low-level primitives.

1.1 Support for Dynamic Execution

The combination of small memory footprint and high adaptivity in software architecture is challenging. On the high end, software component architectures such as CORBA are too heavyweight, as they use the middleware layer on top of an OS to repackage remote procedure calls and to provide a lookup service. On the low end, recent works on firmware update for sensor networks have concentrated on optimizing firmware writing, but they mainly operate on binary images without maintaining a modular structure.

1.2 Memory Optimization

Another key problem is how memory is assigned. In the most general case, all memory is allocated dynamically with malloc() and free() calls, but this approach can be prohibitive for many embedded systems. On the other extreme, such as in TinyOS [8] and others, memory is allocated statically for each routine's own storage. However, static memory allocation can lead to memory fragmentation, since much of the memory may be declared but never fully utilized simultaneously. Although a programmer can manually optimze the memory as overlays, it is not a scalable technique. Moreover, as some program code gets updated, the statically allocated memory locations of unchanged modules may also need to be shifted.

1.3 Scripted Dispatch and MoC-enabled Buffer Optimization

To enable dynamic update of software components, we use a compact dispatcher that invokes native code modules according to a "script." This enables modular update to scripts and native code remotely without reboot. To enable memory optimization without a resident memory manager, we exploit behavioral transparency in models of computation (MoCs) to help optimize memory allocation. A behaviorally transparent model is one whose memory requirements is completely defined by the actors' firing sequence, independent from their actual functionality. This property implies that the memory allocation decision can be made for the entire application prior to execution. One such model is synchronous dataflow (SDF), widely used for modeling and optimization of digital signal processing (DSP) applications.

However, the combination of buffer minimization has to be examined carefully in the context of scripted execution. To maintain modularity, all buffer references could be passed as parameters to each invocation of an actor (i.e., primitive function), but the problem is that the references to the buffers themselves also take up space – as much as the buffers. A more practical way is to allocate buffers contiguously as far as each actor is concerned, so that only a single pointer needs to be passed for each port.

The main contribution of this work is the combination of scripted dispatching structure and buffer minimization as enabled by behavioral transparency. For buffer optimization, we propose a 2-D tile formulation for representing the lifetime of buffers. The algorithm explores alternative schedules to reshape these tiles so that the total memory occupancy is minimized in space and in time. This technique is suitable for scripted execution with minimum overhead on a wide range of embedded systems. The resulting software not only uses minimal memory buffers but also combines the efficiency advantage of statically scheduled code with the dynamic update and modularity advantages of scripting.

We evaluated our techniques with several real-life applications ranging from wearable wireless sensors to networked medical devices. Experimental results show that our implementations satisfy all the code size and memory buffer constraints where previous works have failed, and our solutions also achieve similar runtime efficiency. Our solution will unleash the full potential of these low-power, low-cost embedded systems by enabling them to solve many nontrivial problems using minimal, optimal amounts of resources.

2. BACKGROUND

2.1 Synchronous Dataflow (SDF)

Synchronous dataflow (SDF) is a model of computation for data regular applications including many in DSP [4]. It models computation with a directed graph $G(V,E)$, where the vertices $v \in V$ represent computing processes called *actors*, and the edges $E \subseteq V \times V$ represent *channels* that connect an output port of one actor to an input port of another actor. In general, the actors are *behaviorally transparent* in that when an actor is *fired* (i.e., executed for one iteration), it consumes and produces a known number of tokens as annotated on each of its I/O ports; in SDF, these numbers are constants. These fixed numbers are called *consume rate* and *produce rate*. Each *token* $tk \in E \times \mathbb{N}$ carries data and is uniquely identified by the channel and sequence number. In general, SDF graphs may be cyclic, though we use acyclic ones for the purpose of illustrating behavioral transparency without loss of generality.

Fig. 1 shows an SDF graph with actors A, B, C. Each time A is invoked, it outputs 3 tokens. B consumes 2 and produces 2, and C consumes 1. The channel state of an SDF graph is a vector $\vec{s} = (c_1, \ldots, c_{|E|})^T$ of the token counts on all channels. Firing an actor $v \in V$ will change the state of the SDF graph by adding rate vector $\vec{v}$ that subtracts a fixed token count (i.e., consume rate) from each of the actor's input channels and adds a fixed token count (i.e., produce rate) to each of the actor's output channels. For instance, the state $\vec{s}_i$ can be formed for the two channels (A,B) and (B,C).

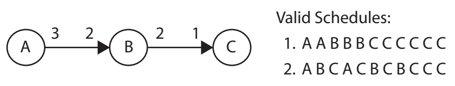


Figure 1: Example SDF graph and its valid schedules

Initially the channels are empty, or $\vec{s}_0 = \binom{0}{0}$. The rate vectors are $\vec{A} = \binom{3}{0}, \vec{B} = \binom{-2}{2}, \vec{C} = \binom{0}{-1}$.

A *schedule* is a sequence of actor firings. An actor v is eligible for firing in state s_i if the next state $\vec{s}_{i+1} = \vec{s}_i + \vec{v}$ does not result in any channel with a token count < 0. Each actor is required to consume and produce tokens in a deterministic sequence, but the timing of their firing is otherwise unconstrained. In this paper, we consider full serialization for the purpose of software implementation. We consider well-posed SDF graphs for which the relative rates can be resolved with finite, bounded buffer space. Because of behavioral transparency, *static scheduling* (or *pre-runtime* scheduling) can be applied efficiently with low runtime overhead by repeatedly invoking the same finite schedule. Different schedules and implementations of runtime support can make a dramatic impact on the code size and buffer size.

2.2 Code Size of SDF Implementations

A *valid* SDF schedule is by definition finite, deadlock-free, and fires each actor at least once. Two valid schedules of the SDF graph shown in Fig. 1 are `AABBBCCCCCC` and `ABCACBCBCCC` (makespan = 11). A compiler could perform code generation by inlining copies of the code to produce a single routine for the whole SDF graph. It has low runtime overhead, but it uses the most code memory. Moreover, if any actor requires update, then the entire routine needs to be replaced, making it costly for in-field update.

One solution to the code size problem is to introduce one level of indirection. A schedule can simply call an actor as a subroutine, though runtime overhead is incurred on the extra level of indirection. One approach that addresses these problems is the *single appearance schedule* (SAS) approach, where each actor appears exactly once, but each may be marked with a number of iterations. For instance, one of the valid schedules in Fig. 1 is `AABBBCCC-CCC`, and it can be converted into an SAS form as `2A3B6C` (execute `A` twice, `B` 3 times, and `C` 6 times). It has reduced code size and simpler dispatch, but the trade-off is that SAS results in larger data buffer requirements, to be discussed in Section 2.3.

2.3 Data Buffer Optimization in SDF

Data memory can be much more constrained than program memory in low-power embedded systems. The program can be stored in either RAM or ROM, and EEPROM and NOR flash may support execute-in-place (XIP) without having to load the program into RAM first. Flash and EEPROM are nonvolatile and do not have the leakage problem that RAM has, but transient data can be allocated only in registers or RAM and thus tend to be much more constrained. Existing techniques that minimize memory for SDF can be classified by their buffer assumptions, into *dedicated buffers*, *shared buffers*, and *merged buffers*.

The *dedicated buffer* approach maps each channel to a dedicated buffer space. This is the default implementation option for SAS and for most implementations that rely on circular buffers. Because SAS decreases code size at the expense of buffer size, several studies attempt to reduce SAS memory by scheduling [5, 12]. Sung [18] and Zitzler [19] relax the assumption by also considering non-SAS for data memory reduction. Oh *et al* combine the benefits of both approaches by representing non-SAS schedules in an SAS form, and expanding it by a dynamic loop mechanism [13].

With *shared buffers*, different channels can share the same buffer space if their tokens' lifetimes do not overlap. In [14], memory is divided into global and local buffers, where local buffers store pointers to a global buffer. Their approach is effective for applications with large streaming data, as in multimedia systems. However, for resource-constrained systems, the pointers themselves can take up as much space as data. Murthy *et al* [10] target shared SAS buffer allocation, though they assume the coarsest granularity, which results in high fragmentation as shown in Section 3. Ritz *et al* [17] present an ILP formulation to minimize buffer size, though their assumptions on flat-SAS (without nested loops) results in large buffer requirement, and their primary objective is code-size reduction.

With *I/O buffer merging*, an actor reuses the input buffer space also for its output buffer. The assumption is that input token is consumed before the output token is generated and no longer needed afterwards [11]. While this enables aggressive buffer sharing, it usually requires additional temporary space to be allocated to each actor. There is no inherent difficulty in applying other techniques with this assumption.

2.4 Runtime Support

Buffer optimization can be effectively applied on adaptive embedded platforms with lightweight runtime support. A recent approach to the runtime support is scripting. ASVM [9] builds an interpretive layer that enables user defined instructions to map to native code, which can correspond to actors. Rappit [7] uses a framework to synthesize the runtime system with a given set of primitives for code size saving. In both cases, a script is more compact than the corresponding compiled binary while retaining the ability for dynamic reconfiguration. However, current scripting schemes have no provisions for buffer memory optimization. Most assume that buffering space is allocated as part of each actor, and the runtime system simply invokes the actor without managing memory. On the other extreme, t-kernel [6] provides virtual memory and SOS [16] provides memory protection for sensor-class microcontrollers, but both incur high runtime overhead and do not exploit knowledge such as SDF firing rates for optimization.

3. ILLUSTRATIVE EXAMPLE

Memory requirements can vary significantly over either different schedules or different buffer allocation schemes for a given schedule. To illustrate these points, consider the SDF graph in Fig. 1.

3.1 Memory Requirements of Different Schedules

For the two valid schedules of the example SDF graph, let us determine the buffer requirement by counting tokens on each channel. We can use a vector of token counts on the channels to represent the state of the SDF graph, as explained in Section 2.1.

The first schedule has the state sequence $\binom{0}{0} \xrightarrow{A} \binom{3}{0} \xrightarrow{A} \binom{6}{0} \xrightarrow{B} \binom{4}{2} \xrightarrow{B} \binom{2}{4} \xrightarrow{B} \binom{0}{6} \xrightarrow{C} \binom{0}{5} \xrightarrow{C,C,C,C,C} \binom{0}{0}$. Therefore, the minimum memory requirement is 6.

The second schedule has the state sequence $\binom{0}{0} \xrightarrow{A} \binom{3}{0} \xrightarrow{B} \binom{1}{2} \xrightarrow{C} \binom{1}{1} \xrightarrow{A} \binom{4}{1} \xrightarrow{C} \binom{4}{0} \xrightarrow{B} \binom{2}{2} \xrightarrow{C} \binom{2}{1} \xrightarrow{B} \binom{0}{3} \xrightarrow{C} \binom{0}{2} \xrightarrow{C} \binom{0}{1} \xrightarrow{C} \binom{0}{0}$. Therefore, the minimum memory requirement is 5 (when the input buffer space is allowed to be reused as output buffer space).

The minimum memory requirement is a lower bound for a schedule, but it is not always achievable by all allocation schemes due to fragmentation or contiguity constraints.

3.2 Buffer Allocation Schemes for a Given Schedule

Fig. 2 shows the solutions generated by different buffer mapping schemes for the same schedule `ABCACBCBCCC`. In each chart, the schedule over time is shown across the top along the X-axis, and the state of the buffer memory cells is shown along the Y-axis. Each data token is labeled by its channel name and its sequence number (e.g., a_1). A blank cell indicates the memory is free; a cell with a token's label means it is being produced; "-" indicates that the token is still alive, and $\rightarrow$ means it is being consumed.

3.3 Coarse-grained, Shared Memory Allocation

Most previous works take the coarsest grain approach, as shown in 2(a). Each channel is allocated a contiguous chunk of memory that is shared between the input and output ports, but different channels do not share space. When A and B fire for the first time, they occupy 3 and 2 units of space, respectively. B also consumes tokens a_1 and a_2, but because a_3 is still unconsumed, a_4, a_5, a_6 must be placed adjacent to a_3 when A fires for the second time (so that a_3, a_4 will be contiguous to B). Hence, six units of space are allocated to channel (A, B). When B fires for the second time, all tokens have been consumed, so tokens b_3, b_4 can reuse the space. However, when B fires for the third time, b_5, b_6 must go into contiguous space, hence bringing (B, C)'s size to 4. The total buffer space is 10, and each firing of A, B, C requires passing 1, 2, 1 pointers, respectively.

3.4 Fine-grained Allocation

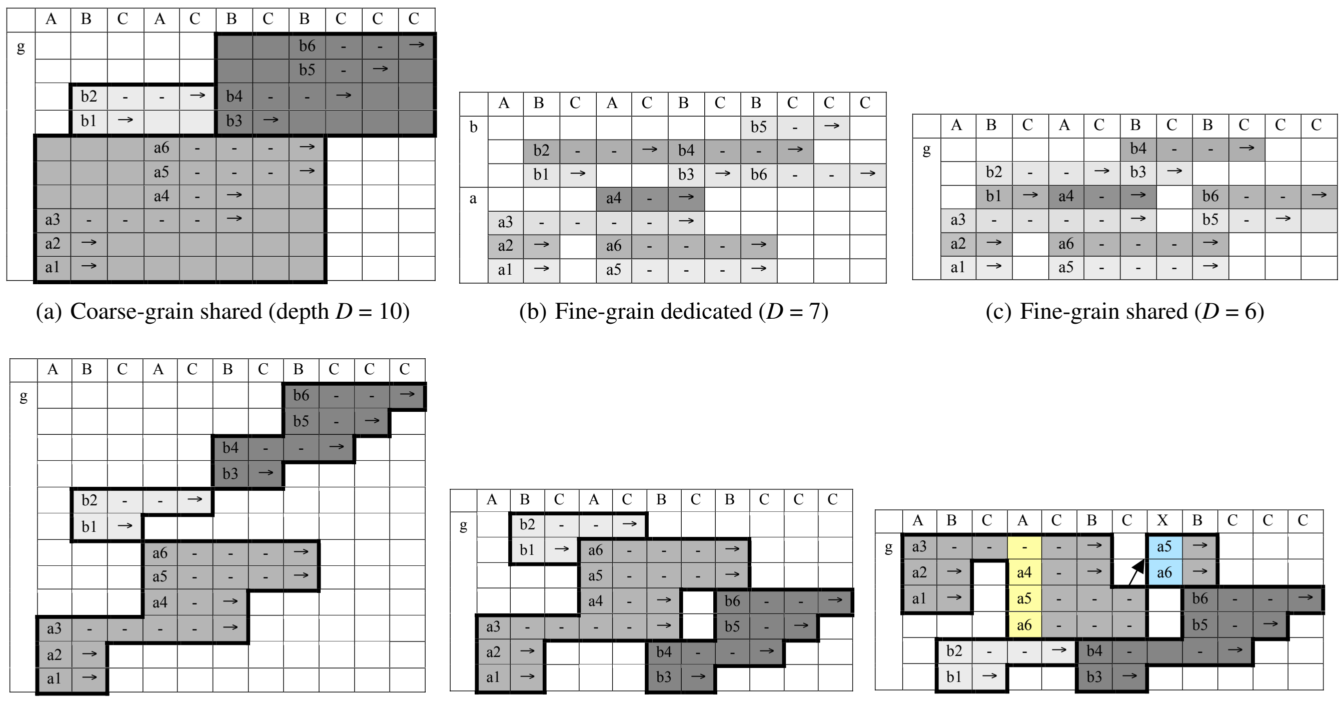
Buffer memory can be reduced by fine-grained allocation, and here we consider dedicated and shared schemes. The former means each channel has its own space, whereas the latter allows different channels to share space. The fine-grained dedicated scheme shown in 2(b) addresses the internal fragmentation problem of 2(a) by using a circular FIFO of size 4 for (A, B) and size 3 for (B, C), thereby cutting the total buffer space from 10 to 7 units. The fine-grained shared scheme shown in 2(c) further eliminates external fragmentation by allowing both channels to allocate space from the same pool, similar to register allocation. It uses 6 units.

However, there is a price to be paid. In the dedicated case, each queue requires a FIFO head and FIFO count, which themselves occupy 2 units of space for each channel and must be referenced by each port. Therefore, the hidden overhead of 2 (head, count) $\times$ 2 channels + 1 (pointer) $\times$ (2 input ports + 2 output ports) = 4 + 4 = 8 total, just to keep track of 7 units of buffer space in the dedicated scheme. In the shared case, each buffer location must be passed to each invocation (or else it must be inlined in the compiled code). A requires 3 pointers on its output port, B requires 2 on input and 2 on output, and C requires 1 pointer on its input. This is 8 additional units of overhead data space for 6 units of buffer space, not including the space taken up by the script.

3.5 Our Approach: Medium-grained, Access-Contiguous Allocation

Our approach is to reduce fragmentation of coarse-grained approaches by shared allocation, and to reduce overhead of fine-grained approaches with contiguous token placement on a per-access basis. We call this medium-grained allocation scheme "access-contiguous" allocation. Figs. 2(d), 2(e), and 2(f) show the three steps that lead to our solution. We start with 2(d), which is similar to 2(a) in respecting the contiguity constraint, but without attempting to greedily reuse the space. Next, 2(e) compacts the memory by rearranging the token-lifetime "islands" in space, similar to the Tetris game, and this cuts the space from 12 downto 7 units.

To achieve even more aggressive mapping, we apply (i) *flip with buffer access direction encoding*, and (ii) *shift with copy* operations, as shown in 2(f). For flip (i), during the second firing of A, instead of placing tokens a_4, a_5, a_6 upwards, we can allocate them downwards so that $a_3 \ldots a_6$ are contiguous – just in the opposite orien-



(a) Coarse-grain shared (depth $D = 10$) (b) Fine-grain dedicated ($D = 7$) (c) Fine-grain shared ($D = 6$)

(d) Medium-grain, step 1 ($D = 12$) (e) Medium, after compaction ($D = 7$) (f) Medium, after flip and shift ($D = 6$)

Figure 2: Possible buffer mapping solutions for a schedule in Fig. 1. The schedule is shown on the top row along x-axis, which denotes time, and the y-axis indicates memory depth.

Table 1: Buffer requirements for different allocation schemes.

Scheme	data buf	buf ptr	head/len.	total
Coase-grained	10	4	0	14
Fine-grained dedic..	7	4	4	15
Fine-grained shared	6	8	0	14
Medium-grained (ours)	6	4	0	10

tation. This can be encoded with a single bit in the pointer. For shift (ii), we effectively defragment the memory by inserting a new actor X to copy a_5, a_6 into the space of a_3, a_4 after they are consumed. This increases the makespan (i.e., schedule length) by 1, but it enables b_1, b_2 to be re-mapped, resulting in 6 units of total buffer space.

In terms of overhead, it uses the same space as the coarse grained approach in that each firing of A, B, C requires passing 1, 2, 1 pointers, respectively. This is the lowest total requirement of all. Table 1 summarizes the total memory usage of each scheme.

4. PROBLEM FORMULATION

This section defines data structures and operators used by our main algorithm for optimizing medium-grained contiguous buffer assignment for SDF schedules. When an actor is fired, it sees each of its input and output ports as a contiguous buffer up to the depth it produces or consumes, though the entire channel need not occupy contiguous space. The objective is to minimize the total memory.

4.1 Data Structures

We now define the symbols and data structures to be used by the main algorithms in the next section.

$FSM(S, \delta, L)$ is a directed graph that captures the *schedule state space*, where S is a set of states as spanned by token-count vectors $\vec{s}$ as shown in Section 3, the state-transition function $\delta \subseteq S \times S$, and the labeling function $L : \delta \rightarrow V$, which maps an edge to an actor.

$G(V, E)$ is a directed graph that captures the *dataflow*, where V represents the set of actors, and E the set of channels.

$\vec{v}$ is an $|E|$-tuple associated with each actor $v \in V$. It indicates the number of tokens to produce (positive) or consume (negative) on each channel each time it is fired.

$tk = (e, k) \in [1, |E|] \times [1, M]$ is called a *token* of sequence number k on channel (edge) numbered e.

ord : $[1, |E|] \times [1, M] \rightarrow \mathbb{Z}^+$ is the *total ordering* function, which maps a token (e, k) to a globally distinct integer in the range $[1, M]$.

M is the *total number of tokens* produced by firing each actor according to a given schedule.

T is the *makespan* of a schedule, namely the number of timesteps.

t_p, t_c : $[1, |E|] \times [1, M] \rightarrow \mathbb{Z}^+$ are the two functions that map a token to the time (step) of its *production* and *consumption*, respectively.

$N = |B_{set}| \in \mathbb{Z}^+$ is the *number of blocks*, as defined by Def. 5.

D called the *memory depth*, is the size of a contiguous region of memory to be allocated to the channels. Note that D is upper bounded by M. D_{opt} is the optimal memory depth.

Definition 1 (Schedule π) is one path of an FSM. $\{\vec{s}_0, \ldots, \vec{s}_T\}$, where $\vec{s}_i \in S$, T = period of a schedule, and $\vec{s}_0 = \vec{s}_T$.

Definition 2 (Schedule Sets S_{all}, S_{set}) S_{all} is set of all possible schedules derived from $G(V,E)$, and S_{set} is a selected subset of S_{all}.

Definition 3 (Token Lifetime Chart *TLC*) is an $M \times T$ matrix whose entries are

$$tlc_{ij} = \begin{cases} tk & \text{if token } tk \text{ is alive at time } j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $i = \text{ord}(tk)$ the total ordering of the token.

Definition 4 (Block *B*) $\subseteq [1,|E|] \times [1,M]$ a set of tokens such that for any two $tk_i, tk_j \in B$, (1) $e_i = e_j = e$, and (2) tk_i overlaps tk_j transitively, that is, if $k_i < k_j$, either

$$\begin{cases} t_p(tk_i) \leq t_p(tk_j) \leq t_c(tk_i) \leq t_c(tk_j) & \text{or,} \\ \text{token } (e, k_i+1) \text{ overlaps } (e, k_j) \text{ transitively} & \text{for } k_i + 1 < k_j \end{cases}$$

Definition 5 (Block Set B_{set}) $\subseteq 2^{[1,|E|]\times[1,M]}$ is the set of all blocks formed by all tokens produced according to a given schedule. $\{B_1,\ldots,B_N\}$, where $B \in B_{set}$ is a block, and $B_i \cap B_j = \emptyset$ if $i \neq j$.

B_{set} serves as the unit entity of the problem. The problem is to correctly place all the block elements in B_{set} inside a given area.

Definition 6 (Block Lifetime Chart *BLC*) is an $N \times T$ matrix where

$$blc_{ij} = \begin{cases} 1 & \text{if block } B_i \text{ contains a live token at timestep } j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Each block is mapped to its lifetime from the earliest produced time to the latest consumed time based on those of its member tokens. That is,

$$t_p(B) = \min_{tk \in B} t_p(tk) \quad (3)$$

$$t_c(B) = \max_{tk \in B} t_c(tk) \quad (4)$$

Definition 7 (Block order vector B_v) $[B_1 \ldots B_N]$ is one permutation of B_{set} for the purpose of determining each block's relative placement in memory. Normally a block with a smaller index is placed first in a lower address, and a subsequent block is placed at the next higher address that does not cause overlap.

Definition 8 (Buffer Map *Bmap*) $D \times T$ matrix where

$$bmap_{ij} = \begin{cases} tk & \text{if token } tk \text{ is mapped to mem. loc. } i \text{ at time } j \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

Each column shows a snapshot of buffer usage while an actor is being invoked. The maximum token count over all columns is the lower bound of data memory requirement.

4.2 Block Operators

To compact memory, we define five operators that move, reorder, and re-shape the blocks: SHIFT, SWAP, FLIP, COPY, and P-FLIP. Fig. 3 illustrates their usage with simple examples.

SHIFT is a basic operator for shifting up or down the block *by a specific offset*. It is normally used to collapse down the upper blocks.

SWAP is to switch the *vertical ordering* of two blocks. Although it is conceptually easy, it can be computationally expensive to switch two blocks, since a straightforward swap may result in intersection with other blocks. For correct swapping, it should follow the sequence shown in Fig. 4.

FLIP is to horizontally flip a block. We classify these shapes and their variations into four types as shown in Fig. 5. Flipping increases the opportunity to compact adjacent blocks.

P-FLIP is to partially flip the block starting from an offset x. It is only allowed when there is only one input token at the flip-point. This is especially useful for Type4 shapes as in Fig. 3, since the staircase-shaped blocks tend to cause fragmentation and prevent other types of optimizations.

COPY is to insert a simple actor that copies input data to a specified output location for the purpose of defragmentation. It can reduce the memory depth but at the expense of extra execution delay. It can be applied optionally for more aggressive optimizations.

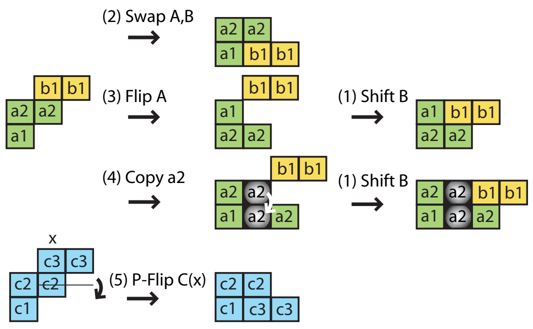


Figure 3: Illustration of the five operators for the buffer layout algorithm.

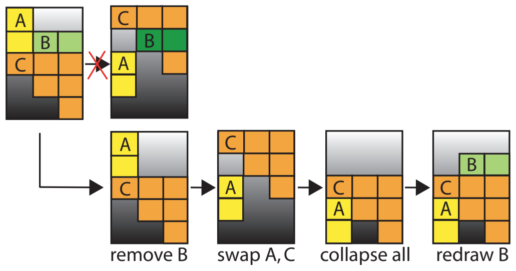


Figure 4: Example of SWAP operation: a direct switch (block A and C) can be invalid due to conflicts with other blocks (block B), and in general requires a full sequence to operate.

5. BUFFER OPTIMIZATION ALGORITHMS

This section first defines the top-level steps for memory optimization and introduces the core algorithms for buffer mapping.

5.1 Top-level Flow

Fig. 6 captures the flow of our memory optimizing approach. Input to the host is an SDF graph and output is a generated script that

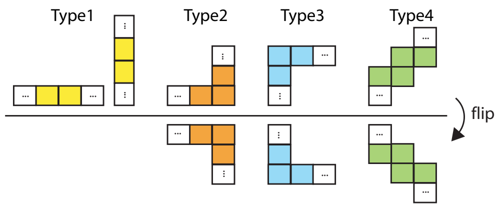


Figure 5: Possible shapes of blocks

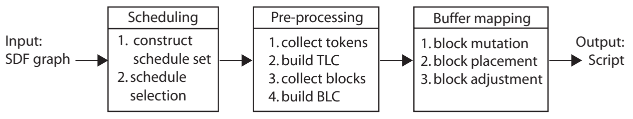


Figure 6: Top-level flow of our memory optimization

```
SCHEDULE SELECTION (SDF graph $G(V,E)$)
    Phase I: Build $FSM$
    $Q$.push($\vec{s}_0$)                          ▷ $Q$ is queue of next states to visit
    while $Q$ do
        for each $v \in (V - \{v_{src}\})$ do          ▷ try firing all nonsrc actors
            if $v$ eligible to fire then              ▷ $\exists$ enough tk to consume
                $\vec{s}_{next} \leftarrow \vec{s}_{cur} + \vec{v}$
                $FSM$.addTransition($v$, $\vec{s}_{next}$)      ▷ add transition and state
                $Q$.push($\vec{s}_{next}$)
        if not ($\exists$ fired $v$) do               ▷ if necessary, fire source actor
            $\vec{s}_{next} \leftarrow \vec{s}_{cur} + \vec{v_{src}}$
            $FSM$.addTransition($v$, $\vec{s}_{next}$)          ▷ add transition and state
            $Q$.push($\vec{s}_{next}$)
    Phase II: Collect all schedules
    for each path $\in FSM$ do
        if not ($\exists \vec{s}_{next}$) or ($\vec{s}_{next} = \vec{s}_0$) then
            $S_{all} \leftarrow S_{all} \cup \{path\}$      ▷ add path to the all schedule set
    Phase III: Select schedules
    for each $\pi$ in $S_{all}$ do
        if $D_{opt} \neq \max(|\vec{s}|)$ then           ▷ rm opt-unreachable schedules
            $S_{set} \leftarrow S_{all} - \pi$
    if $|S_{set}| > maxNS$ then              ▷ max Number of $\pi$ is configurable...
        sort $S_{set}$ in the incr. order of the longest subsequence of $v$
        $S_{set} \leftarrow S_{set}[1 \ldots maxNS]$
    return $S_{set}$
```

Figure 7: The Schedule Selection Algorithm.

contains the final schedule and buffer map. Each line of script includes an actor and i/o pointers. The script is dispatched to remote node at runtime and is executed according to its order.

The first step is to obtain the schedule set and select best candidate schedules. The second step is the preparation for the next step, by obtaining the building blocks and data structures. Finally, buffer mapping is performed by mutation, placement, and adjustment of the blocks for each schedule. The following sections present detailed algorithms for these steps.

5.2 Schedule Selection Algorithm

The idea of the SCHEDULE SELECTION algorithm is to build and prune a finite state machine (FSM) that effectively captures the *states of schedules* for a given SDF graph. To recall, the state can be represented as a vector $\vec{s}_i$ that indicates the number of tokens on each channel. The algorithm constructs an FSM that connects these $\vec{s}$ vectors in breadth-first (BFS) order, and the transitions are labeled with the actor whose firing causes the change of state. This way, a sequence of T path labels along this FSM starting from v_{src} corresponds to an actual schedule for the given SDF graph. The advantage to using the FSM is that it is a compact representation that captures all possible schedules of a given SDF graph, without having to explicitly enumerate all schedules. Moreover, it also enables very effective pruning during the construction of the FSM, thereby enabling selection of the optimal (or a near-optimal) schedule without state explosion.

The SCHEDULE SELECTION algorithm is shown in Fig. 7. It consists of three phases: I. Build the FSM, II. Collect the schedules, and III. Select schedules. While choosing which next state of the FSM to add or traverse, we use a greedy heuristics to help prune the solution space. It chooses those states that have the minimum

	Phase	operators
1	Initialization	-
2	Mutation	FLIP
3	Placement a. EPF b. LBF c. BOA	SWAP, SHIFT
4	Adjustment	P-FLIP, COPY

(a)

Level	Sequence of phases
Default	$1 \rightarrow 3a$
Opt-1	$1 \rightarrow 3(ab)$
Opt-2	$1 \rightarrow 3(abc)$
Opt-3	$1 \rightarrow 3(abc) \rightarrow 4$
Opt-4	$1 \rightarrow 2 \rightarrow 3(abc) \rightarrow 4$
Opt-5	$1 \rightarrow 2 \rightarrow 3(abc) \rightarrow 4 \rightarrow$ $1 \rightarrow 2 \rightarrow 3(abc) \rightarrow \ldots$

(b)

Figure 8: Buffer mapping steps and level-specific sequences

of the maximum number of live tokens. The number of live tokens can be computed by summing values of a state vector $\vec{s}_i$. The maximum value reached during an entire schedule execution is a lower bound on the memory buffer depth (D_{opt}), although it is not always achievable due to fragmentations. In addition, the configuration parameter *maxNS* can be used to set a threshold on the maximum number of schedules to consider. When the number of schedules exceeds this threshold, another heuristics can be triggered for a different level of pruning. Another heuristic is to order the schedules in increasing order of *length of consecutive subsequence* of an actor. For example, `AABAA` is preferred over `AAAAB`, as it enables more modular blocks and leads to less fragmentation, as shown in Section 7.

The worst case runtime complexity of the algorithm is $O(|V| \times |S|)$. *FSM* state space $|S|$ itself can be exponential in general, but it can be pruned during construction using the greedy heuristic and the *maxNS* threshold as suggested above, in addition to others such as maximum channel depth and the maximum number of live tokens while choosing the next state vector to add. Also, during Phase II, a greedy heuristic can prune paths that violate certain criteria (e.g., monotonic increase of live tokens beyond a threshold length).

5.3 Buffer Layout Algorithm

Among the selected candidate schedules, we iterate through each and apply the Buffer Layout algorithm to find the best solution. Fig. 8(a) shows the four phases of the algorithm. These phases are applied in different combinations based on the optimization level chosen. We provide one default level plus five optimization levels, as shown in Fig. 8(b). Each level progressively reduces the buffer requirement while taking more time to compute the layout.

Initialization From schedule set S_{set}, retrieve one π and perform the following sequence:

step	time complexity
Obtain *TLC*	$O(M)$
Collect *B*	$O(M)$
Obtain *BLC*	$O(T \times M)$
Collect blocks' 2-D information (borders, shape, size, etc)	$O(T \times M)$

Mutation is performed by selectively flipping a set of blocks, based on the shapes of the intersecting blocks (i.e., blocks with overlapping lifetimes).

Placement is the actual block layout. One possible solution is to layout the initial configuration, and continuously perform SWAP and SHIFT to reach a more compact layout. However, as shown in Fig. 4, the operations may be costly. Alternatively, we can explore in a bottom-up manner by choosing block orders that build up a compact layout. A solution is derived from a block order B_v, dropping each block B_i from top to the lowest possible memory location. The key to the optimal solution is to find the best B_v. Three ways of obtaining B_vs are 1) Earliest Produce-time First (EPF), 2) Largest Block-size First (LBF), and 3) Block Order Algorithm

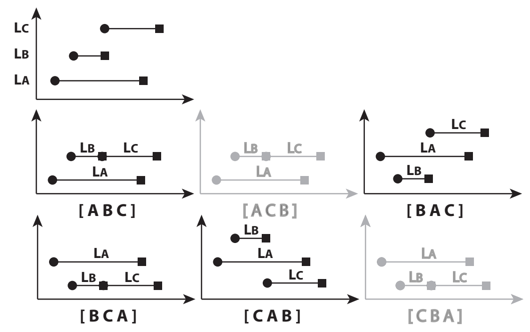


Figure 9: Example of redundant block orders (in gray). Top: *BLC* of Fig. 2(d); Lower: possible block orderings.

```
BOA(BLC)
$t_0 \leftarrow 1$
for $t \leftarrow 2$ to $T$ do                          ▷ get sliced intervals $iv \in Iv$
    for each $B \in Bset$ do
        if $(t = B_{start})$ then                        ▷ start a new interval
            $iv \leftarrow (t_0, t)$                         ▷ iv is an interval
            $Iv \leftarrow Iv \cup iv$                       ▷ Iv is an interval set
            $t_0 \leftarrow t$
sort $Iv$ in increasing order of $iv_{start}$
for each $iv \in Iv$ do
    $t_0 \leftarrow iv_{start}$
    $intvBset \leftarrow$ [blocks with $t_0$]         ▷ set of blocks starting at $t_0$
    if 1st iteration then                    ▷ no fixed blocks at the beginning...
        $slots \leftarrow \{1\}$                     ▷ thus, only one slot available
    else
        $slots \leftarrow$ all slots between fixed blocks
    for each $B$ in $intvBset$          ▷ constructing all paths to min $B_v$ set
        for each slot in $slots$
            $partial\_B_v \leftarrow$ computed placement
            $partial\_B_vset \leftarrow partial\_B_vset \cup \{partial\_B_v\}$
    select a $partial\_B_v$ to fix                     ▷ for the next interval
Build min($B_vset$)                                   ▷ walk through partial_b_vset
```

Figure 10: The Block-ordering Algorithm.

(BOA). 1) and 2) are straightforward, and Section 5.4 explains 3). **Adjustment** attempts further optimization by P-FLIP and COPY.

5.4 Block-ordering Algorithm (BOA)

Block-ordering Algorithm (BOA) prunes out all the redundant B_v's and provides minimum B_v set. The redundancy of exhaustive search is illustrated in Fig. 9 with example *BLC* (Block Lifetime Chart) derived from Fig. 2(d). Although there are six possible B_v's, two B_v's ($[ACB], [CBA]$) are redundant, since they point to already existing solutions.

Fig. 10 shows the algorithm. Minimum B_v set is obtained by chronological relative ordering. Note that we are considering only the lifetimes and vertical ordering of blocks without the 2-D knowledge. Fig. 11 illustrates the algorithm with the same example. The first step is to slice the whole period into smaller intervals $iv \in Iv$. We have three intervals in this example, and the borders are where each block's lifetime starts. For each iv, we first collect vertical slots that indicate the relative orders the block can choose from. They are spaces between already fixed blocks. By placing each block in each slot, we derive partial B_vs, which are paths to obtaining the final B_v set. We fix the block to one slot before proceeding to the next interval. It is shown that the output B_v set obtained this way is the same as the one in Fig. 9 without redundancy.

BOA reduces the worst-case runtime complexity from $O(N!)$ downto $O(N \times I_{avg})$, where I_{avg} is the average number of intersect-

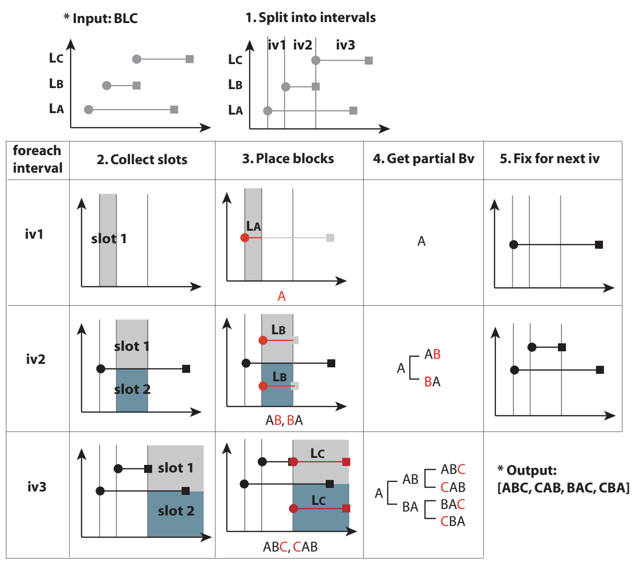


Figure 11: Illustration of Block Ordering Algorithm (BOA). L_B is block B's lifetime and shaded area indicates current slots for each time interval.

ing blocks at each time unit. For the example in Fig. 14(a), the solution space is pruned from 362,880 to 4,374.

6. DISPATCHING SCHEME

The dispatcher is part of an embedded OS being developed by the authors. Its details are outside the scope of this paper, and here we summarize the part relevant to buffer allocation. The buffer-optimized result from Section 5 is formed as a script and then transferred and loaded into the embedded system. It specifies a sequence of actors to fire in the following format:

actor address	repeat factor	input ptr(s)	output ptr(s)

Actor address is the pointer to the function that implements an actor, *repeat factor* is the number of consecutive iterations of an actor instance, and *input ptr(s)* and *output ptr(s)* are the pointers to input/output buffers respectively, and could be more than one, depending on fan-in/out property of an actor.

The script dispatcher simply fetches an actor address from the script memory and jumps to the address. The following fields (e.g., repeat facter, etc) are fetched locally by the actor, whose format conforms to the script interface. The idea is based on threaded-code [3], but our dispatcher is much simpler in that it fetches only operators (i.e., actors), and arguments are automatically fetched and used by the operators. The behavioral transparency is exploited via the actors' predefined memory usage. The trade off of the reduced dispatch overhead is the offline effort to write actor functions in a memory-explicit fashion and to format the script at the host side.

7. EXPERIMENTS

The techniques described in this paper have been implemented and tested over a suite of applications on a number of resource-constrained platforms. This section describes the experimental setup, applications, and experimental results.

7.1 Tools and Experimental Platforms

We implemented our SDF modeling, schedule selection, and buffer optimization algorithms in an integrated tool, whose GUI front-end is shown in Fig. 12. It visualizes the memory map and memory usage profile. The tool also produces a text report which contains

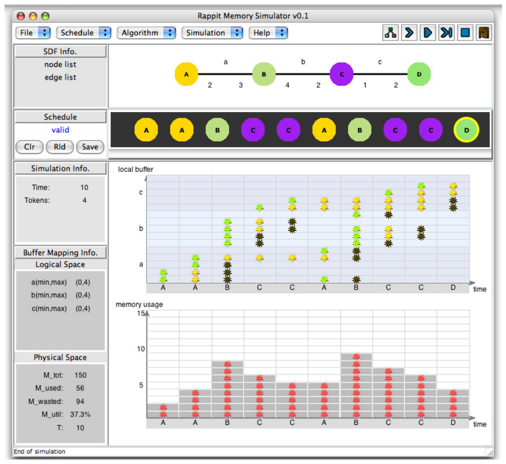


Figure 12: GUI for modeling, scheduling, buffer optimization.

layout information as in Fig. 16. Our experimental results are generated by this tool. In addition, the back-end of this tool is integrated into our development framework named Rappit [7], which supports host-assisted, interactive environment for reprogramming and rapid prototyping embedded systems. In this environment, both firmware and application code can be loaded to a remote system at runtime. The firmware is compiled by SDCC [1], and the application is composed as a script by our utility tools.

We ported the applications onto several sensor platforms, including Eco [15] and Atmel's AVR Butterfly [2] shown in Fig. 13. Eco contains 4KB RAM and 4KB EEPROM. The AVR Butterfly contains 1KB RAM, 512B EEPROM, 16KB program flash, and 512KB external data flash, and its built-in sensors include temperature/light sensors, and 3-axis accelerometer (Eco). Other peripherals on the AVR butterfly include an SD (Secure Digital) Card interface through SPI, an LCD screen, a speaker (PWM output), and a joystick.

Sample SDF applications are shown in Fig. 14. The first is the filter bank [10]. We also have three wireless sensor applications, including 3-axial accelerometer to RF, wireless receiver, and a wireless data logger with a secure digital (SD) card and an LCD. These benchmarks are of representative complexity for ultra-compact wireless sensor nodes and many other deeply embedded systems. The SDF graphs may appear simple, but the state space explodes quickly in naive implementations. Automation is needed because we expect the user to load new SDF graphs as script interactively, and thus the host computer must perform on-the-fly scheduling and buffer allocation before transmitting the schedule to the embedded system. Fig. 13(e) shows the interactive environment, where host wirelessly communicates with the sensor node through a base station node, which is connected to host via serial port.

7.2 Results

7.2.1 *Impact of Schedule Selection*

The total buffer savings depend on both scheduling and buffer mapping performance. Fig. 15 reveals the importance in scheduling. Each schedule imposes its own D_{opt} value, and they can vary in a wide range. For instance, the schedule of Fig. 15(a) cannot reach $D < 27$ due to the large block sizes and shapes. An alternative schedule, which generates more modular blocks, shown in Fig. 15(b), enables much better compaction, with $D = 17$. There-

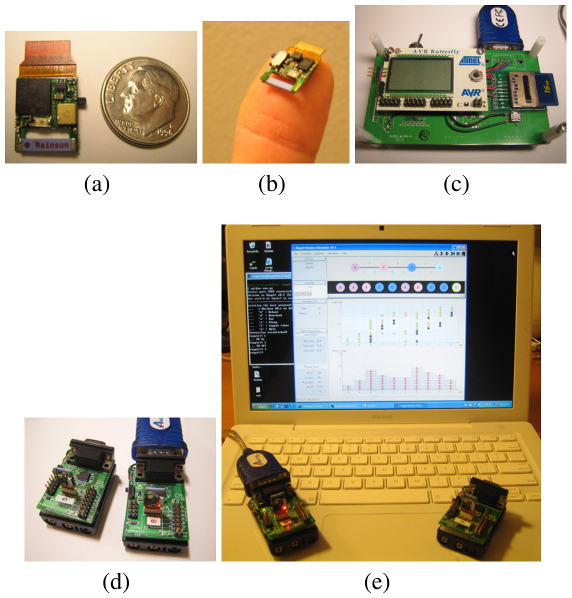


Figure 13: Target platforms and experimental setup. (a) Eco wireless sensor node (b) Eco on a finger (c) AVR Butterfly (d) Eco as sensor node and base station (e) Integrated host/node scripting environment

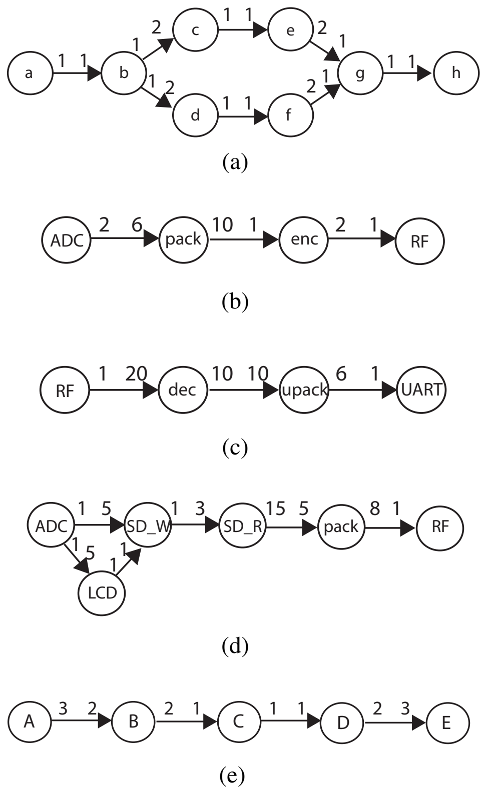
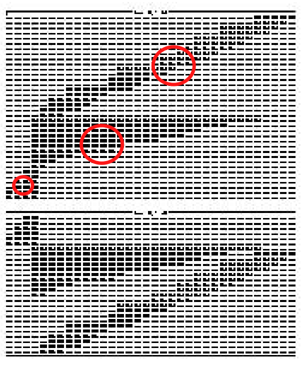


Figure 14: SDF graphs used in experiments. (a) Depth-1 filter bank (b) 3-axis acceleration sensor (c) Wireless receiver (d) RF data logger w/ SD card, LCD (e) Synthetic


(a)

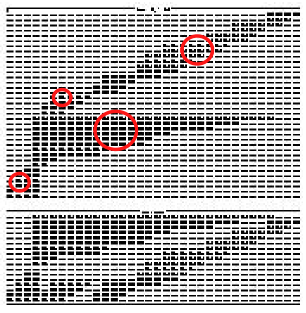

(b)

Figure 15: Different amounts of buffer savings enabled by different schedules. Top graphs start from $D = 37$ and bottom graphs are the optimized results. (a) A schedule with 3 blocks, reached $D = 27$. (b) A schedule with 4 blocks, reached $D = 17$.

fore, it is useful to generate or select schedules that compose a number of modular blocks. More modular blocks are achievable by controlling the actors' firing in a way that keeps the number of net tokens low, so that no channel buffers tokens for too long. Such direction highly improves the performance of our heuristics, since a good schedule set prunes out unnecessary iterations for buffer mapping and helps derive an optimal schedule much faster.

7.2.2 *Convergence of Heuristics*

Fig. 16 displays an automatically generated output of our algorithm for the input filter bank application whose SDF graph is shown in Fig. 14 (a). The blocks are highlighted in different shades. It starts from a straightforward initial allocation, and then applies GREEDY FIT, SWAP, and PFLIP-AND-COPY. Each step shows gradual reduction of memory depth and fragmentation (denoted as F) until the final result reaches the optimal buffer depth $D_{opt} = 5$. It shows in detail how blocks are transformed and reordered to find an efficient layout in a 2-D space.

Fig. 17 shows the collected results of buffer saving steps. For each SDF graph, the amount of buffer depth D is shown along buffer reduction steps (1 to 5). Each line in a graph is a result of a different schedule. Some show results of less number of schedules since its SDF permits only a few possible schedules (e.g., one schedule for Fig. 17(c)). It is shown that different schedules show slightly different saving patterns, but as the algorithm reaches the last step (block adjustment with P-FLIP and COPY) they all reach optimal solution. Note that each step further causes more execution delay during interactive execution, and thus a user may choose to stop at step 3 or 4. It is also observed that each schedule not only sets the lower bound of possible savings as shown in Section 7.2.1, but also shows how many iterations it takes to reach D_{opt}.

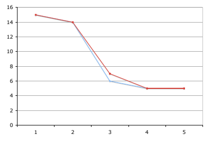
7.2.3 *Memory Savings*

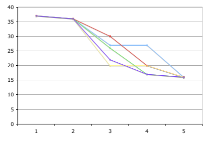
Table 2 shows the results of buffer size reduction. The baseline is unshared buffer, where each channel has its own space, and this is how most SDF implementations work. Here we present our results based on medium-grain assumptions, where the contiguity constraint applies. Our level-3 optimization achieves an average memory saving of 53.6% while incurring similar overhead.

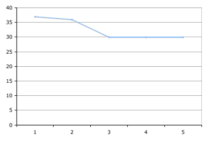
Table 3 compares the overhead incurred by fine-grained vs. medium-grained. The former requires that every buffer location be tracked and becomes expensive quickly with the number of tokens pro-

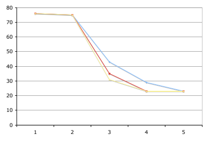
```
******************* Depth = 14, F =  66 *******************
---- ---- ---- ---- ---- ---- ---- ---- ---- GH02 GH02 GH02
---- ---- ---- ---- ---- ---- ---- ---- GH01 GH01 GH01 ----
---- ---- ---- ---- ---- ---- ---- EG02 EG02 EG02 ---- ----
---- ---- ---- ---- ---- ---- ---- EG01 EG01 ---- ---- ----
---- ---- ---- ---- ---- ---- CE01 CE01 ---- ---- ---- ----
---- ---- ---- ---- ---- FG02 FG02 FG02 FG02 FG02 ---- ----
---- ---- ---- ---- ---- FG01 FG01 FG01 FG01 ---- ---- ----
---- ---- ---- ---- DF01 DF01 ---- ---- ---- ---- ---- ----
---- ---- AB02 AB02 ---- ---- ---- ---- ---- ---- ---- ----
---- ---- ---- BC02 BC02 BC02 BC02 ---- ---- ---- ---- ----
---- BC01 BC01 BC01 BC01 BC01 BC01 ---- ---- ---- ---- ----
---- ---- ---- BD02 BD02 ---- ---- ---- ---- ---- ---- ----
---- BD01 BD01 BD01 BD01 ---- ---- ---- ---- ---- ---- ----
AB01 AB01 ---- ---- ---- ---- ---- ---- ---- ---- ---- ----

******************* Depth = 13, F =  58 *******************
---- ---- ---- ---- ---- ---- ---- ---- ---- GH02 GH02 GH02
---- ---- ---- ---- ---- ---- ---- ---- GH01 GH01 GH01 ----
---- ---- ---- ---- ---- ---- ---- EG02 EG02 EG02 ---- ----
---- ---- ---- ---- ---- ---- ---- EG01 EG01 ---- ---- ----
---- ---- ---- ---- ---- ---- CE01 CE01 ---- ---- ---- ----
---- ---- ---- ---- ---- FG02 FG02 FG02 FG02 FG02 ---- ----
---- ---- ---- ---- ---- FG01 FG01 FG01 FG01 ---- ---- ----
---- ---- AB02 AB02 DF01 DF01 ---- ---- ---- ---- ---- ----
---- ---- ---- BC02 BC02 BC02 BC02 ---- ---- ---- ---- ----
---- BC01 BC01 BC01 BC01 BC01 BC01 ---- ---- ---- ---- ----
---- ---- ---- BD02 BD02 ---- ---- ---- ---- ---- ---- ----
---- BD01 BD01 BD01 BD01 ---- ---- ---- ---- ---- ---- ----
AB01 AB01 ---- ---- ---- ---- ---- ---- ---- ---- ---- ----

******************* Depth =  8, F =  26 *******************
---- ---- ---- ---- ---- ---- CE01 CE01 ---- ---- ---- ----
---- ---- ---- ---- ---- ---- ---- EG02 EG02 EG02 ---- ----
---- ---- ---- ---- ---- ---- ---- EG01 EG01 GH02 GH02 GH02
---- ---- AB02 AB02 DF01 DF01 ---- ---- GH01 GH01 GH01 ----
AB01 AB01 ---- BD02 BD02 FG02 FG02 FG02 FG02 FG02 ---- ----
---- BD01 BD01 BD01 BD01 FG01 FG01 FG01 FG01 ---- ---- ----
---- ---- ---- BC02 BC02 BC02 BC02 ---- ---- ---- ---- ----
---- BC01 BC01 BC01 BC01 BC01 BC01 ---- ---- ---- ---- ----

*********************** Memory Layout ***********************
 A    B    A    B    D    F    C    E    G    G    H    H
******************* Depth =  5, F =   4 *******************
---- ---- ---- BC02 BC02 BC02 BC02 EG02 EG02 EG02 ---- ----
---- BC01 BC01 BC01 BC01 BC01 BC01 EG01 EG01 ---- ---- ----
---- ---- ---- BD02 BD02 FG02 FG02 FG02 FG02 FG02 ---- ----
---- BD01 BD01 BD01 BD01 FG01 FG01 FG01 FG01 GH02 GH02 GH02
AB01 AB01 AB02 AB02 DF01 DF01 CE01 CE01 GH01 GH01 GH01 ----
***********************************************************
```

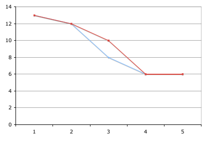
Figure 16: Result of filter bank example


(a)


(b)


(c)


(d)


(e)

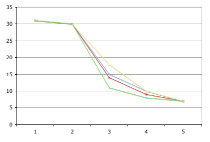

(f)

Figure 17: Convergence of buffer savings. x-axis denotes the optimization level, y-axis denotes memory depth. Each line indicates a fixed schedule. (a) filter bank (b) sensor (c) receiver (d) data logger (e) synthetic1 (f) synthetic2

Table 2: Buffer size comparison. Opt-1 is greedy fit; Opt-2 is swap; Opt-3 is pflip and copy;

Application	Fig#	Ded	Sh	Opt 1	Opt 2	Opt 3	% sav.
filter bank	14(a)	14	13	7	5	5	64.3
sensor	14(b)	36	35	21	16	16	55.6
receiver	14(c)	36	35	29	29	29	19.4
data logger	14(d)	76	75	29	23	23	69.7
synthetic1	1	12	11	9	7	6	50.0
synthetic2	14(e)	30	29	9	9	7	60.0
geometric mean							53.6

Table 3: Memory requirements of medium vs. fine grained with bookkeeping overhead.

	Fine grained			Medium grained			%sav.
Application	buf	ptr	tot	buf	ptr	tot	
filter bank	5	20	25	5	16	21	16.0
sensor	16	23	39	16	6	22	43.6
receiver	29	48	77	29	6	35	54.5
data logger	23	47	70	23	12	35	50.0
synthetic1	6	8	14	6	4	10	28.6
synthetic2	7	15	22	7	8	15	36.8
geo. mean							40.2

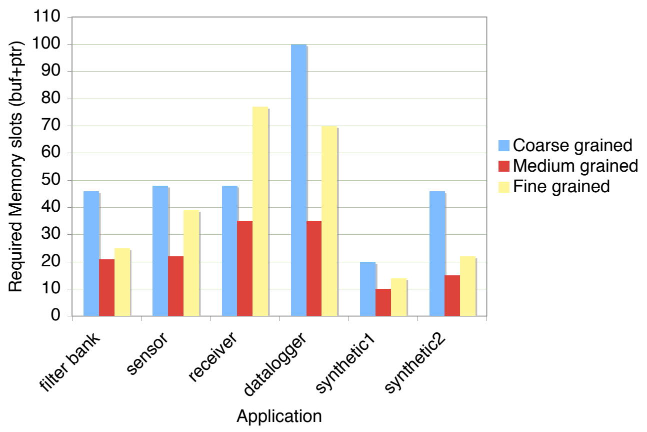


Figure 18: Memory requirements (buffer+pointer) of different applications by different granularities. Medium grained consistently shows high savings.

duced or consumed on each firing. The latter requires 2× the number of channels independent of the number of tokens and thus scale much better. We achieve an average reduction of 40.2% over fine grained solutions when bookkeeping overhead is accounted for.

Fig. 18 shows the final result of buffer requirement including the bookkeeping overhead. Although coarse-grained approach normally shows largest requirement due to its worst-case buffer assignment, in some cases, fine-grained result appears to be the worst (e.g., receiver). It is due to the high bookkeeping overhead. Our medium grained approach consistently shows higher savings than the other approaches.

8. CONCLUSION

This paper presents a new buffer optimization scheme combined with a script dispatching structure for memory-constrained, behaviorally transparent embedded systems. It saves code size by scripting, and exploits the regularity of dataflow models for memory buffer optimization. Our medium, "access-contiguous" buffer granularity effectively reduces fragmentation often seen with coarse grained approaches, without the high overhead of fine-grained approaches. Combined with our lightweight dispatching mechanism, the buffer optimization is effectively applied to modular embedded systems with minimal overhead. It enables emerging ultra-compact, highly adaptive embedded platforms to fully utilize the precious memory without paying a high price for the abstraction. As memory is currently the limiting factor, our technique has shown to be an enabling technology for a new class of real-world, deeply embedded applications.

9. REFERENCES

[1] SDCC – Small Device C Compiler. `http://sdcc.sourceforge.net/`.

[2] Atmel Corporation. In *http://www.atmel.com/*.

[3] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[4] S. Bhattacharyya, P. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell MA, 1996.

[5] S. Bhattacharyya, P. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *DAES'97*, 2:33–60, January 1997.

[6] L. Gu and J. A. Stankovic. t-kernel: Providing reliable os support for wireless sensor networks. In *Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*, November 2006.

[7] J. Hahn, Q. Xie, and P. H. Chou. Rappit: framework for synthesis of host-assisted scripting engines for adaptive embedded systems. In *(CODES+ISSS'05)*, pages 315–320, September 2005.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[9] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[10] P. Murthy and S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *(TCAD'01)*, 20(2):177–198, April 2001.

[11] P. Murthy and S. Bhattacharyya. Buffer merging – a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM TODAES*, 9(2):212–237, April 2004.

[12] P. K. Murthy, S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Journal of Formal Methods in Systems Design*, 11(1):41–70, July 1997.

[13] H. Oh, N. Dutt, and S. Ha. Shift buffering technique for automatic code synthesis from synchronous dataflow graphs. In *(CODES+ISSS'05)*, pages 51–56, September 2005.

[14] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs for multimedia applications. In *(DAC'02)*, pages 275–280, 2002.

[15] C. Park and P. H. Chou. Eco: Ultra-wearable and expandable wireless sensor platform. In *Third International Workshop on Body Sensor Networks (BSN'06)*, April 2006.

[16] R. Rengaswamy, E. Kohler, and M. B. Srivastava. Harbor: Software-based memory protection for sensor nodes. In *6th International Symposium on Information Processing in Sensor Networks*, April 2006.

[17] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 2651–2654, May 1995.

[18] W. Sung and S. Ha. Memory efficient software synthesis using mixed coding style from dataflow graph. *TVLSI*, 8(1):522–526, October 2000.

[19] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Evolutionary algorithm based exploration of software schedules for digital signal processors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1762–1770, 1999.