Journal of Signal Processing Systems 50, 163–177, 2008
© 2007 Springer Science + Business Media, LLC. Manufactured in The United States.
DOI: 10.1007/s11265-007-0114-1

Memory-constrained Block Processing for DSP Software Optimization

MING-YUNG KO
Sandbridge Technologies Inc., White Plains, NY, USA

CHUNG-CHING SHEN AND SHUVRA S. BHATTACHARYYA
Electrical and Computer Engineering Department, University of Maryland, College Park, MD, USA

Received: 21 February 2007; Revised: 20 May 2007; Accepted: 13 June 2007

Abstract. Digital signal processing (DSP) applications involve processing long streams of input data. It is important to take into account this form of processing when implementing embedded software for DSP systems. Task-level vectorization, or block processing, is a useful dataflow graph transformation that can significantly improve execution performance by allowing subsequences of data items to be processed through individual task invocations. In this way, several benefits can be obtained, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSP oriented addressing modes. On the other hand, block processing generally results in increased memory requirements since it effectively increases the sizes of the input and output values associated with processing tasks. In this paper, we investigate the memory-performance trade-off associated with block processing. We develop novel block processing algorithms that carefully take into account memory constraints to achieve efficient block processing configurations within given memory space limitations. Our experimental results indicate that these methods derive optimal memory-constrained block processing solutions most of the time. We demonstrate the advantages of our block processing techniques on practical kernel functions and applications in the DSP domain.

Keywords: block processing, dataflow, embedded systems, vectorization, block diagram compiler, activation overhead, context switch, memory management

1. Introduction

Indefinite- or unbounded-length streams of input data characterize most applications in the digital signal processing (DSP) and communications domains. As the complexity of DSP applications grows rapidly, great demands are placed on embedded processors to perform more and more intensive computations on these data streams. The multi-dimensional requirements that are emerging in commercial DSP products—including requirements on cost, time-to-market, size, power consumption, latency, and throughput—further increase the challenge of DSP software implementation.

Because of the intensive, stream-oriented computational structure of DSP applications, performance optimization for DSP software is a widely researched area. Examples of methods in this area include reducing context switching costs, replacing costly instructions that use absolute addressing, exploiting specialized hardware units or features, and using various other DSP-oriented compiler optimization techniques (e.g., see [10]).

Task-level *vectorization* or *block processing* is one general method for improving DSP software performance in a variety of ways. In this context, block processing refers to the ability of a task to process groups of input data, rather than individual scalar data items, on each task activation. Such a task is typically implemented in terms of a block processing parameter that indicates the size of the each input block that is to be processed. This way, the task programmer can optimize the internal implementation of the task through awareness of its block processing capability, and a task-level design tool can optimize the way block processing is applied to each task and coordinated across tasks for more global optimization.

In this paper, we explore such global block processing optimization for dataflow-based design tools. Due to its intuitive match to signal flow graphs, and its capabilities for improving verification and optimization, dataflow is becoming increasingly popular as the semantic basis for DSP-oriented languages and tools. Commercial examples of tools that provide dataflow-based design capability for DSP include ADS by Agilent Technologies, LabVIEW by National Instruments, and Cocentric System Studio by Synopsys. Relevant research tools include DIF [6], PeaCE [15], Ptolemy [3], and StreamIt [16].

More specifically, in this paper, we examine the trade-off between block processing implementation and data memory requirements. Understanding this trade-off is useful in memory constrained software and design space exploration. Theoretical analysis and algorithms are proposed to efficiently achieve streamlined block processing configurations given constraints on data memory requirements. In addition, our approach is based on hierarchical loop construction such that code size is always minimized (i.e., duplicate copies of actor code blocks are not required). Experimental results show that our approach often computes optimal solutions. At the same time, our approach is practical for incorporation into software synthesis tools due its low polynomial run-time complexity.

A preliminary version of this work has been published in [7]. In this version, we go beyond the preliminary version in a number of significant ways. First, we explore a more generalized block processing formulation that can handle arbitrary blocking factors; the version in [7] is restricted to the conventional case of unity blocking factor. This is an important generalization since the blocking factor is generally an important parameter in regards to overall vectorization performance. In addition, an algorithmic framework is proposed in the paper to integrate our block processing algorithm with arbitrary techniques for reducing data memory requirements. Several theoretical developments that support the validity and efficiency of our algorithms are introduced in this paper as well.

2. Related Work

To strengthen the motivation for block processing, it has been shown that block processing improves regularity and thus reduces the effort in address calculation and context switching [4]. Block processing also facilitates efficient utilization of pipelines for vector-based algorithms, which are common in DSP applications [2].

Task-level, block processing optimization for DSP was first explored by Ritz et al. [14]. In this approach, a dataflow graph is hierarchically decomposed based on analysis of fundamental cycles. The decomposition is performed carefully to avoid deadlock and maximize the degree of block processing. While the work jointly optimizes block processing and code size, it does not consider data memory cost. Another limitation of this approach is its high complexity, which results from exhaustive search analysis of fundamental cycles.

Joint optimization of block processing, data memory minimization, and code size minimization is examined in [13]. Unlike the approach of [14], the work of [13] employs memory space sharing to minimize data memory requirement. However, again the techniques proposed are not of polynomial complexity, and therefore they may be unsuitable for large designs or during design space exploration, where one may wish to rapidly explore many different design configurations.

In both the methods of [13] and [14], the optimization problem is formulated without user-specified data memory constraints. Furthermore, although, overall memory sharing cost is minimized in [13], memory costs for individual program variables are fixed to be the largest. In fact, however, many configurations of program variable sizes—in particular, the sizes of buffers that implement the edges in the dataflow graph—are usually possible under dataflow semantics. Choosing carefully within this space of buffer configurations leads to smaller memory requirements and provides flexibility in memory cost tuning.

In contrast to these related efforts, the optimization problem that we target in this paper is formulated to take into account a user-defined data memory bound.

This corresponds to the common practical scenario where one is trying to fit the implementation within a given amount of memory (e.g., the on-chip memory of a programmable digital signal processor). Also, by iterating through different memory bounds, trade-off curves between performance and memory cost can be generated for system synthesis and design space exploration.

In this paper, in conjunction with block processing optimization, memory sizes of dataflow buffers are efficiently configured through novel algorithms that frequently achieve optimum solutions, while having low polynomial-time complexity.

Various other methods address the problem of minimizing context switching overhead when implementing dataflow graphs. For example, the *retiming* technique is often exercised on single-rate dataflow graphs. In the context of context switch optimization, retiming rearranges delays (initial values in the dataflow buffers) so they are better concentrated in isolated parts of the graph [8, 17]. As another example, Hong et al. [5] investigate throughput-constrained optimization given heterogeneous context switching costs between task pairs. The approach is flexible in that overall execution time or other objectives (such as power dissipation) are jointly optimized under a fixed schedule length through appropriate sequencing of task execution.

These efforts target different objectives and operate on *single-rate* dataflow graphs, which are graphs in which all task execute at the same average rate. In contrast, the methods targeted in this paper operate on *multirate* dataflow graphs, which are common in many signal processing applications, including wireless communications, and multimedia processing. Our work is motivated by the importance of multirate signal processing, and the much heavier demands on memory requirements that are imposed by multirate applications.

3. Background

In the DSP domain, applications are often modeled as *dataflow* graphs, which are directed graphs in which nodes (*actors*) represent computational tasks and edges represent data dependences. FIFO buffers are usually allocated to hold data as it is transferred across the edges. *Delays*, which model instantiations of the Z^{-1} operator, are also associated with edges, and are typically implemented as initial data values in the associated buffers.

For DSP system implementation, it is often important to analyze the memory requirements associated with the FIFOs for the dataflow edges. In this context, the *buffer cost* (or *buffer size*) of a buffer means the maximum amount of data (in terms of bytes) that resides in the buffer at any instant.

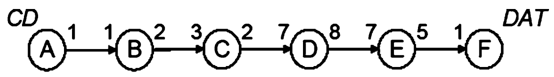
For DSP design tools, *synchronous dataflow (SDF)* [9] is the most commonly used form of dataflow. SDF imposes the restriction that for each edge in the dataflow graph, the numbers of data values produced by each invocation of the source actor and the number of data values consumed by each invocation of the sink actor are constant values. Given an SDF edge e, these production and consumption values are represented by $prd(e)$ and $cns(e)$, respectively, and the source and sink actors of e are represented by $src(e)$ and $snk(e)$, respectively.

A *schedule* is a sequence of actor invocations (or *firings*). We compile an SDF graph by first constructing a *valid schedule*, which is a finite schedule that fires each actor at least once, and does not lead to unbounded buffer accumulation (if the schedule is repeated indefinitely) nor buffer underflow (deadlock) on any edge. To avoid buffer overflow and underflow problems, the total amount of data produced and consumed is required to be matched on all edges. In [9], efficient algorithms are presented to determine whether or not a valid schedule exists for an SDF graph, and to determine the minimum number of firings of each actor in a valid schedule. We denote the *repetitions count* of an actor as this minimum number of firings, and we collect the repetitions counts for all actors in the *repetitions vector*. The repetitions vector is indexed by the actors in the SDF graph and it is denoted by q. Given an SDF edge e and the repetitions vector q, the *balance equation* for e is written as

$$q(src(e))prd(e) = q(snk(e))cns(e).$$

To save code space, actor firings can be incorporated within loop constructs to form *looped schedules*. Looped schedules group sequential firings into schedule loops; each such loop is composed of a loop iteration count and one or more iterands. In addition to being actor firings, iterands can also be subschedules, and therefore, it is possible to represent nested-loop constructs.

The notation we use for a schedule loop L is $L = (nI_1I_2 \ldots I_m)$, where n denotes the iteration count and

a


b $q(A, B, C, D, E, F) = (147, 147, 98, 28, 32, 160)$

c $SAS = ((49(3AB)(2C))(4(7D)(8E(5F))))$

Figure 1. **a** An SDF graph modeling of CD to DAT rate conversion. **b** The repetitions vector. **c** A SAS.

$I_1, I_2, \ldots, I_m$ denote the iterands of L. *Single appearance schedules (SASs)* are schedules in which each actor appears only once. In inlined code implementation, an SAS contains a single copy of code for every actor and results in minimal code space requirements. Figure 1 is drawn to demonstrate an SDF graph representation of *CD (compact disc)* to *DAT (digital audio tape)* sample rate conversion, along with the associated repetitions vector, and one possible SAS for this application.

Any SAS can be transformed to an *R-schedule*, $S = (i_L S_L)(i_R S_R)$, where $S_L (S_R)$ is the left (right) subschedule of S [1]. The binary structure of an R-schedule can be represented efficiently as a binary tree, which is called a *schedule tree* or just a *tree* in our discussion [11]. In this tree representation, every node is associated with a loop iteration count, and every leaf node is additionally associated with an actor. A schedule tree example is illustrated in Fig. 2.

The loop hierarchy of an R-schedule can easily be derived from a schedule tree, and vice versa. Therefore, R-schedules and schedule trees are referred to interchangeably in our work.

To avoid confusion when referring to terms for schedule trees and SDF graphs, some conventions are introduced here. When referring to general graph structure terms, such as "node" and "edge", we refer to these terms in the context of schedule trees, unless otherwise specified.

Given a node a, the left (right) child is denoted as $left(a)$ ($right(a)$). We define the *association operator*, denoted $\alpha()$, as follows: $\alpha(A) = a$ maps the SDF actor A to its associated schedule tree leaf node a. The loop iteration count associated with a leaf node a is denoted as $l(a)$. The tree (or subtree) rooted at node r is denoted as $tree(r)$, and the corresponding set of internal nodes (the set of nodes in $tree(r)$ that are not leaf nodes) is represented as $\lambda(r)$ or $\lambda(tree(r))$.

4. Block Processing Implementation

When a large volume of input data is to be processed iteratively by a task, a block processing implementation of the task can provide several advantages, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSP-oriented addressing modes. Motivation for block processing is elaborated on qualitatively in [14]. In this section, we add to this motivation with some concrete examples.

An example of integer addition is given in Fig. 3 to illustrate the difference between conventional (scalar) and block processing implementation of an actor. From the perspective of the *main()* function, function *add_vector()* in Fig. 3b has less procedure call overhead, fast addressing through auto-increment modes, and better locality for pipelined execution compared to *add_scalar()* in Fig. 3a.

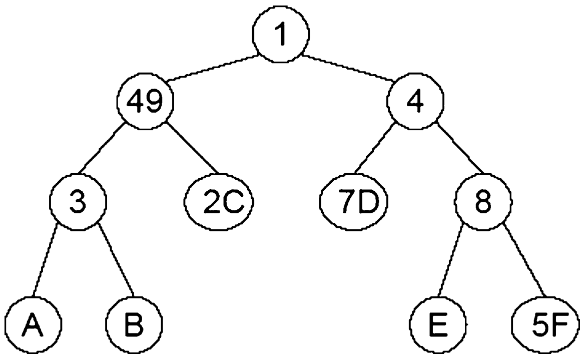


Figure 2. A schedule tree example corresponding to the SAS of Fig. 1c.

a
```
void add_scalar(int a, int b, int* sum) {
      *sum = a + b;
}
main() {
      int[] arrayA, arrayB, arraySum;
      for (int i=0; i<size; i++)
            add_scalar(arrayA[i], arrayB[i],
                         &arraySum[i])
}
```

b
```
void add_vector(int* a, int* b, int* sum,
            int size) {
      int* a2=a, b2=b, sum2=sum;
      for (int i=0; i<size; i++)
            *sum2++ = (*a2++) + (*b2++);
}
main() {
      int[] arrayA, arrayB, arraySum;
      add_vector(arrayA, arrayB, arraySum,
            size);
}
```

Figure 3. Integer addition **a** scalar version, **b** vector version.

To further illustrate the advantages of block processing, different configurations of a *convolution* actor, which is an important DSP kernel function, are evaluated on the Texas Instruments TMS320C6700 processor. The results are summarized in Fig. 4. Here, the left chart shows the number of execution cycles versus the number of actor invocations for both scalar and block processing implementations, where in the block processing case, all actor invocations are achieved with a single function invocation. The right chart gives the execution cycles reduced through application of block processing. Lines are drawn between dots to interpolate the underlying trend and do not represent real data.

By inspecting these charts, block processing is seen to achieve significant performance improvement, except when the actor invocation count (*vectorization degree*) is unity. In this case, one must pay for the overhead of block processing without being able to amortize the overhead over multiple actor invocations, so there is no improvement. Moreover, improvements are seen to saturate for sufficiently high vectorization degrees.

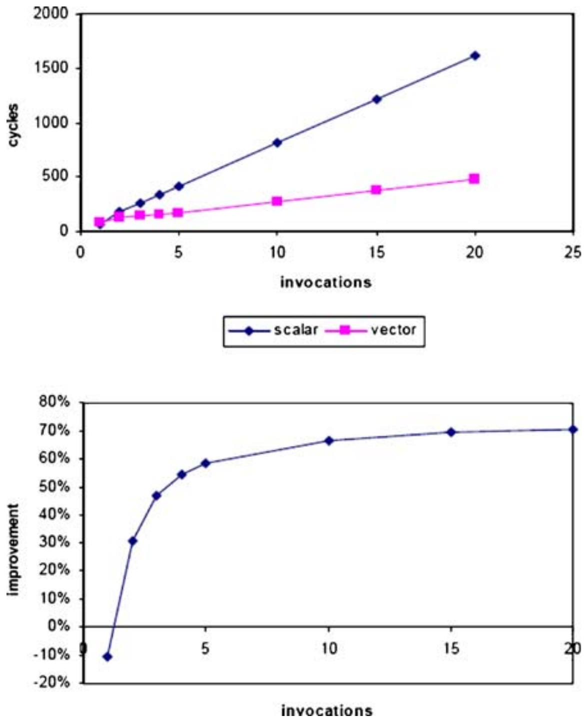


Figure 4. Performance comparison of vectorized and scalar implementation of convolution operation.

Charts of this form can provide application designers and synthesis tools helpful quantitative data for applying block processing during design space exploration.

5. Block Processing in Software Synthesis

To model block processing in SDF-based software synthesis, we convert successive actor invocations to inlined loops embedded within a procedure that represents an activation of the associated actor. Here, the number of loop iterations is equivalent to the number of successive actor invocations—that is, to the vectorization degree. Given an actor A, we represent the vectorization degree for A in a given block processing configuration as $vect(A)$. Thus, each time A is executed, it is executed through a unit of $vect(A)$ successive invocations. This unit is referred to as an *activation* of A.

Under block processing, the number of data values produced or consumed on each activation of A is $vect(A)$ times the number of data values produced or consumed per invocation of A (as represented by the $prd(e)$ and $cns(e)$ values on the edges that are incident to A).

Useful information pertaining to block processing can be derived from schedule trees. For this purpose, we denote $act(r)$ as the *activations number* for $tree(r)$ rooted at r. This quantity is defined as follows: $act(r) = 1$ if r is a leaf node, and otherwise, $act(r) = l(r)(act(left(r)) + act(right(r)))$.

If r is a leaf node and $\alpha(R) = r$, then $vect(R)$ successive invocations of actor R are equivalent to a single activation, and $act(r) = 1$. If r is an internal node, then based on the structure of SASs, an activation is necessary when $left(r)$ completes, and is followed by $right(r)$ at each of $l(r)$ iterations. An activation occurs also when $right(r)$ completes in one iteration and is followed by $left(r)$ in the next iteration. Therefore, we have $l(r)(act(left(r)) + act(right(r)))$ activations for $tree(r)$.

Given a valid schedule S of an SDF graph, there is a unique positive integer $J(S)$ such that S invokes each actor A exactly $J \times q(A)$ times, where q is the repetitions vector, as defined in Section 3. This positive integer is called the *blocking factor* of the schedule S. The blocking factor can be expressed as

$$J(S) = gcd(inv(A_1), inv(A_2), \ldots, inv(A_n)),$$

where gcd represents the *greatest common divisor* operation, $inv(A_i)$ represents the number of times that actor A_i is invoked in the schedule S, and n is the number of actors in the given SDF graph. We say that S is a *minimal* valid schedule if $J(S) = 1$, and a *graph iteration* corresponds to the execution of a minimal valid schedule, or equivalently, the execution of $q(A)$ invocations of each actor A.

Increasing the blocking factor beyond the minimum required value of 1 can reduce the overall rate at which activations occur. For example, suppose that we have a minimal valid schedule $S_1 = ((2A)(3B))$ and another schedule $S_2 = ((8A)(12B))$, which has $J = 4$. Although both schedules result in 2 activations, the average rate of activations (in terms of activations per graph iteration) in schedule S_2 is one-fourth that of S_1. This is because S_2 operates through four times as many graph iterations as schedule S_1.

As motivated by this example, we define the *activation rate* of a schedule S as $rate(S) = act(S)/J(S)$, where $act(S)$ is the total number of actor activations in schedule S.

If S is represented as $tree(r)$, we have

$$rate(S) = rate(tree(r)) = act(r)/J(S).$$

The problem of optimizing block processing can then be cast as constructing a valid schedule that minimizes the activation rate. For example, S_2 has a lower activation rate ($rate(S_2) = 0.5$) than S_1 ($rate(S_1) = 1$), and S_2 is therefore more desirable under the minimum activation rate criterion.

The blocking factor is closely related to, but not equivalent to, the *unfolding factor*. Unfolding is a useful technique in DSP dataflow graph analysis [12], and both the unfolding factor and blocking factor are intended to help in investigating hardware/software configurations that encapsulate more than one graph iteration. While unfolding makes multiple copies of the original actors to enhance execution performance (in a manner analogous to loop unrolling), and generally allows executions of multiple graph iterations to overlap, use of blocking factors does not imply any duplication of actors, and usually does imply that each iteration of the given schedule will execute to completion before the next iteration begins.

6. Activation Rate Minimization with Unit Blocking Factor (ARMUB)

In this section we consider in detail the problem of minimizing the activation rate in a manner that takes into account user-defined constraints on buffer costs (data memory requirements). We restrict ourselves to unit blocking factor here because we are interested in memory-efficient block processing configurations, and increases in blocking factor generally increase memory requirements [1]. The resulting optimization problem, which we call *ARMUB* (Activation Rate Minimization with Unit Blocking factor), is the problem of rearranging the schedule tree of a minimal valid schedule such that the resulting schedule is valid and has a minimum number of total activations.

Two more restrictions in these formulations are that the input SDF graph is assumed to be acyclic, and the edges are assumed to be delayless. Acyclic SDF graphs represent a broad and important class of DSP applications (e.g., see [1]). Furthermore, through the loose interdependence scheduling framework [1], which decomposes general SDF graphs into hierarchies of acyclic SDF graphs, the techniques of this paper can be applied also to general SDF graphs. The assumption of delayless edges is done mainly for simplicity and clarity in the presentation. Our methods can easily be extended to handle nonzero delays, and we outline such extension at the end of this section.

The problem is formally described as follows. Assume that we are given an acyclic SDF graph G and a valid schedule S (and associated schedule tree $tree(r)$) for G such that $J(S) = 1$. Block processing is to be applied to G by rearranging the loop iteration counts of tree nodes in the schedule tree for S. The optimization variables are the set $\{l(x)\}$ of loop iteration counts in the leaf nodes of the rearranged schedule tree (recall that these loop iteration counts are equivalent to the vectorization degrees of the associated actors). The objective is to minimize the number of activations:

$$min(act(r)). \quad (1)$$

Changes to the loop iteration counts of tree nodes must obey the constraint that the overall numbers of actor invocations in the schedule is unchanged. In other words, for each SDF actor A,

$$q(A) = \prod_{x \in path(a,r)} l(x), \quad (2)$$

where $\alpha(A) = a$, and $path(a, r)$ is the set of nodes that are traversed by the path from the leaf node a to the root node r. Intuitively, the equation says that no matter how the loop iteration counts are changed along the path, their product has to match the repetitions count of the associated actor.

In the ARMUB problem, we are also given a buffer cost constraint M (a positive integer), such that the total buffer cost in the rearranged schedule cannot exceed M. That is,

$$\sum_{e} buf(e) \leq M, \tag{3}$$

where $buf(e)$ denotes the buffer size on SDF edge e.

The structure of R-schedules permits efficient computation of buffer costs.

Theorem 1 Given an acyclic SDF graph edge e, we have two leaf nodes a and b associated with the source and sink: $\alpha(src(e)) = a$ and $\alpha(snk(e)) = b$. Let p be the least common parent of a and b in the schedule tree. Then the buffer cost on e can be evaluated by the following expressions.

$$buf(e) = prd(e)\left(\prod_{x \in path(a, left(p))} l(x)\right) = cns(e)\left(\prod_{y \in path(b, right(p))} l(y)\right). \tag{4}$$

Proof To determine the buffer cost, we need to figure out when data is produced and consumed on e. According to the semantics of single appearance schedules, at each of the $l(p)$ iterations of p, data produced on e by $left(p)$, which involves the firing of actor $src(e)$, will not be consumed until $right(p)$ (which involves the firing of actor $snk(e)$) starts to execute. In addition, based on the balance equations and the assumption of delayless edges, all of the data produced throughout a given iteration of $l(p)$ will be consumed without any data values remaining on the edge for the next iteration. This identical production and consumption pattern recurs at all $l(p)$ iterations in $tree(p)$ and any subtree where $tree(p)$ is contained. Therefore, the buffer cost is equivalent to the amount of data produced or consumed (Eq. (4)). □

In summary, the ARMUB problem can be set up by casting Eq. (1) through Eq. (4) into *a non-linear programming (NLP)* formulation, where the objective is given by Eq. (1), the variables are the loop iteration counts of the schedule tree nodes, and the constraints are given in Eqs. (2), (3), and (4). Due to the intractability of NLP, efficient heuristics are desired to tackle the problem for practical use.

To determine an initial schedule to work on, we must consider the potential optimization conflicts between buffer cost and activations. While looped schedules that make extensive use of nested loops are promising in generating low buffer costs, activations minimization favors *flat schedules*, that is, schedules that do not employ nested loops.

We employ nested-loop SASs that have been constructed for low buffering costs as the initial schedules in our optimization process because flat schedules can easily be derived from any such schedule by the setting loop iteration counts of all internal nodes to one, while setting the loop iteration counts of leaf nodes according to the repetitions counts of the corresponding actors. Furthermore the construction of buffer-efficient nested loop schedules has been studied extensively, and the results of this previous work can be leveraged in our approach to memory-constrained block processing optimization. Specifically, the *APGAN (Acyclic Pairwise Grouping of Adjacent Nodes)* and *GDPPO (Generalized Dynamic Programming Post Optimization)* algorithms are employed in this work to compute buffer-efficient SASs as a starting point for our memory-constrained block processing optimization [1].

6.1. Loop Iteration Count Factor Propagation

As described earlier, activation values of leaf nodes are always equal to one and independent of their loop iteration counts. Hence, one approach to optimizing activations values is to enlarge the loop iteration counts of leaf nodes by absorbing the loop iteration counts of internal nodes. A similar approach is proposed in [14] to deal with cyclic SDF graphs with delays. The strategy in [14] is to extract integer factors out of a loop's iteration count and carefully propagate the factors to inner loops. Propagations are validated by checking that they do not introduce deadlock. However, as described in Section 2, the work of [14] does not consider buffer cost in the optimization process.

For acyclic SDF graphs, as we will discuss later, factors of loop iteration counts should be aggressively propagated straight to the inner most iterands to achieve effective block processing. Under memory constraints, such propagation should be balanced carefully against any increases in buffering costs.

Definition 1 *(FActor Propagation toward Leaf nodes (FAPL))* Given $tree(r)$, the FAPL operation, when it can be applied, is to extract an integer factor $V > 1$ out of $l(r)$ and merge V into the loop iteration counts of all the leaf nodes in $tree(r)$. Formally, the new loop iteration count of r is $l(r)/V$, and for every leaf f in $tree(r)$ the new loop iteration count is $V \cdot l(f)$. Loop iteration counts of internal nodes remain unchanged. For notational convenience, a FAPL operation is represented as $\phi(r, V)$ for $tree(r)$ and factor V, or simply as ϕ when the context is known. We call r the 'FAPL target internal node', all leaf nodes in $tree(r)$ the 'FAPL target leaf nodes', and V the 'FAPL factor'.

An example of FAPL is illustrated in Fig. 5. FAPL reduces the number of activations and increases buffer costs.

Theorem 2 Given $tree(r)$ with $act(r)$, $\phi(r, V)$ reduces the activations of $tree(r)$ by a factor of V.

Proof From the definitions of $act()$ and FAPL, $act(left(r))$ and $act(right(r))$ are not affected (remain unchanged) from the operation $\phi(r, V)$. On the other hand, the loop iteration count of r turns into $l(r)/V$ as a result of $\phi(r, V)$. Therefore, the new activations, $act'(r)$, are updated as

$$act'(r) = \frac{l(r)}{V}(act(left(r)) + act(right(r))) = \frac{act(r)}{V}.$$ □

Definition 2 Given an SDF edge e with $\alpha(src(e)) = a$, and $\alpha(snk(e)) = b$, we call $\phi(r, V)$ an 'effective FAPL' on e if $a, b \in \lambda(r)$. Conversely, we call e a 'APL effective edge' from ϕ.

Theorem 3 Given an SDF edge e and an effective FAPL $\phi(r, V)$ on e, the FAPL increases the buffer cost on e by a factor of V.

Proof Suppose that $\alpha(src(e)) = a$, and $\alpha(snk(e)) = b$. With $\phi(r, V)$, we obtain new loop iteration counts $l'(a) = V \cdot l(a)$, $l'(b) = V \cdot l(b)$ for a, b, respectively. Suppose that w is the smallest parent of a and b. Then w must be contained in $tree(r)$. Along the paths of $path(a, left(w))$ and $path(b, right(w))$, the loop iteration counts of all the internal nodes of $tree(w)$ remain unchanged. Therefore, from Theorem 1, we can conclude that the buffer cost on e is increased by a factor of V. □

Theorem 4 Given a valid schedule, the new schedule that results from a FAPL operation is also a valid schedule.

Proof First, a FAPL operation changes loop iteration counts only and in particular, such an operation does not change the topological sorting order associated with the initial schedule. Therefore, neither data dependencies nor schedule loop nesting structures are affected by FAPL operations. Given a FAPL operation effective on an SDF edge e, the same numbers of data values (Theorem 1) will be produced and consumed on the buffer associated with e and no buffer underflow nor overflow problems will be incurred. □

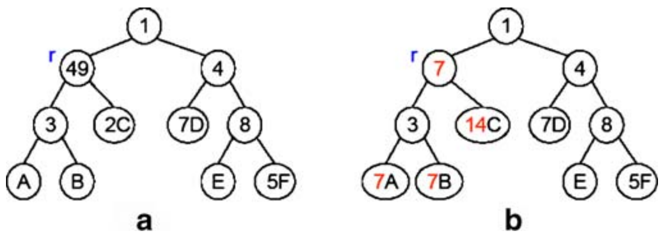


Figure 5. With $\phi(r, 7)$, the tree in **a** turns to that in **b**.

6.2. Properties of FAPL Sequences

We call a sequence $\phi_1 \cdot \phi_2 \cdot \ldots \cdot \phi_m$ of FAPL operations applied sequentially (from left to right) as a *FAPL sequence*, and we represent such a sequence compactly by $\prod \phi_i$.

Definition 3 Given two FAPL sequences,

$$\Phi_1 = \prod_{i=1\ldots m} \phi_{1,i}$$

and

$$\Phi_2 = \prod_{j=1\ldots n} \phi_{2,j}$$

we say that Φ_1 and Φ_2 are 'equivalent' (denoted as $\Phi_1 \leftrightarrow \Phi_2$) if for every tree node a of $tree(r)$, we have $l_1(a) = l_2(a)$, where l_1 and l_2, respectively, give the loop iteration counts of tree nodes for the schedules that result from Φ_1 and Φ_2.

There are some useful properties about FAPL sequences, which we state here: *commutativity* ($(\phi_1 \cdot \phi_2) \leftrightarrow (\phi_2 \cdot \phi_1)$) and *associativity* ($((\phi_1 \cdot \phi_2) \cdot \phi_3) \leftrightarrow (\phi_1 \cdot (\phi_2 \cdot \phi_3))$).

Theorem 5 $(\phi_1 \cdot \phi_2) \leftrightarrow (\phi_2 \cdot \phi_1)$.

Proof Suppose that $\phi_1 = \phi(r_1, V_1)$ and $\phi_2 = \phi(r_2, V_2)$. For any leaf node a and $a \in \lambda(r_1) \cap \lambda(r_2) \neq \emptyset$, the loop iteration count $l'(a)$ is equal to $l(a)V_1V_2$. If $r_1 = r_2$, the new loop iteration count, $l'(r_1)$, is updated as $l(r_1)/(V_1V_2)$. □

Theorem 6 $((\phi_1 \cdot \phi_2) \cdot \phi_3) \leftrightarrow (\phi_1 \cdot (\phi_2 \cdot \phi_3))$.

Proof Suppose that $\phi_1 = \phi(r_1, V_1)$, $\phi_2 = \phi(r_2, V_2)$, and $\phi_3 = \phi(r_3, V_3)$. First, the right hand side can be reformatted as $(\phi_2 \cdot \phi_3) \cdot \phi_1$ according to the commutativity property provided in Theorem 5.

For any leaf node a such that $a \in \lambda(r_1) \cap \lambda(r_2) \cap \lambda(r_3) \neq \emptyset$, the new loop iteration count, $l'(a)$, $l(a)V_1V_2V_3$. If $r_1 = r_2 = r_3$, the new loop iteration count, $l'(r_1)$, is updated as $l(r_1)/(V_1V_2V_3)$. If $a \in \lambda(r_1) \cap \lambda(r_2)$ and $a \notin \lambda(r_3)$, updating of the new loop iteration count $l'(a)$ is equivalent to $\phi_1 \cdot \phi_2 = \phi_2 \cdot \phi_1$. Similarly, if $a \in \lambda(r_1) \cap \lambda(r_3)$ and $a \notin \lambda(r_2)$, updating of $l'(a)$ is equivalent to $\phi_1 \cdot \phi_3 = \phi_3 \cdot \phi_1$.

Analogous reasoning applies to the updated value of $l'(r_1)$ if $r_1 = r_2 \neq r_3$ or if $r_1 = r_3 \neq r_2$. For the conditions of $a \in \lambda(r_2) \cap \lambda(r_3)$, $a \notin \lambda(r_1)$, or $r_2 = r_3 \neq r_1$, updating of the new loop iteration counts $l'(a)$ and $l'(r_2)$, is equivalent to that of $\phi_2 \cdot \phi_3$.

In summary, in all of the cases examined above, $(\phi_1 \cdot \phi_2) \cdot \phi_3 \leftrightarrow (\phi_2 \cdot \phi_3) \cdot \phi_1$ holds. □

6.2.1. FAPL-based Heuristic Algorithms. The problem of FAPL-based activations minimization is complex due to the interactions between the many underlying optimization variables. In this section, we propose an effective polynomial-time heuristic, called *GreedyFAPL*, for this problem.

GreedyFAPL, illustrated in Fig. 6, performs block processing from pre-computed integer factorization results for the loop iteration counts of internal nodes. First, internal nodes are sorted in decreasing order based on their activations values. The sorted internal nodes are traversed one by one as FAPL target internal node targets. For each internal node target, the integer factors of the loop iteration count are tried in decreasing order as the FAPL factors until a successful FAPL operation (i.e., a FAPL operation that does not overflow the given memory constraint *M*) results. It is based on this consideration of integer factors in decreasing order that we refer to GreedyFAPL as a greedy algorithm.

Because the schedule tree is a binary tree and all leaf nodes are associated with unique actors, there are $|V|$ iterations in the outer *for* loop to traverse the internal nodes, where V is the set of actors in the input SDF graph. In the buffer cost constraint validation, we do not need to recompute the total buffer cost each time. Only the increase in buffer cost needs to be determined because we keep track of the overall buffer cost at each step.

In the worst case, for a given FAPL operation, all members of the edge set E in the input SDF graph are the FAPL-effective edges. Hence, the buffer cost constraint validation takes $O(|E|)$ time for each iteration of the inner *for* loop of GreedyFAPL. For every FAPL operation, updating the loop iteration counts of the target leaf nodes, and evaluating the

```
Algorithm: GreedyFAPL
Input: An SDF graph, $tree(r)$, and buffer cost upper bound $M$
Output: Minimum activations

sort internal nodes by decreasing activations
for (each internal node $a$) {
    sort integer factors of $l(a)$ decreasingly
    for (each factor $\pi$ of $l(a)$) {
        compute overall buffer cost, $C$, as if $\phi(a, \pi)$ was run
        if ($C \leq M$) {
            execute $\phi(a, \pi)$
        }
    }
}
return $act(r)$ as output
```

Figure 6. The GreedyFAPL algorithm for the ARMUB problem.

buffer costs of the FAPL-effective SDF edges costs involves $O(|V| + |E|)$ run-time complexity.

If Ω is the maximum number of factors in the prime factorization of the loop iteration count of an internal node, then the computational complexity of GreedyFAPL can be expressed as $O(\Omega|V|(|V| + |E|))$.

6.3. Handling of Nonzero Delays

As mentioned early in this section, the techniques developed here can easily be extended to handle delays. Specifically, given an SDF edge e with D units of delay, and a FAPL operation on e, the buffer cost induced from a FAPL operation on e can be determined as the sum of

$$D + F_{delayless}, \tag{5}$$

where $F_{delayless}$ represents the buffer cost of a FAPL operation on the "delayless version of" e—that is, the SDF edge obtained from e by simply changing the delay to zero. By using this generalized buffer cost calculation throughout (observe that when $D = 0$, the sum in Eq. (5) still gives the correct result, from the definition of $F_{delayless}$), the methods developed earlier in this section can be extended to general acyclic SDF graphs (i.e., graphs with arbitrary edge delays). A similar extension can be performed for the techniques developed in the following section, but again for clarity, we will develop the technique mainly in the context of delayless graphs.

7. Activation Rate Minimization with Arbitrary Blocking Factor (ARMAB)

In this section, we generalize the problem formulation of the previous section to include consideration of non-unity blocking factors, called *ARMAB* for brevity. The resulting figure of merit of activation rate, $rate(S)$, provides an approximate measure to model the overall block processing performance enhancement of a schedule. In other words, schedules that have lower activation rates generally result in schedules with better performance. In Section 8, we include experimental results that quantify this claim that activation rate is a good indicator of overall schedule performance.

In this section we build on the insights developed in the previous section. Note that the problem we target in this section is strictly more general than that targeted in the previous section since ARMUB, as defined in Section 6, has the constraint $J(S) = 1$. Our study of the more restricted problem in Section 6 resulted in useful insights and techniques, most notably the GreedyFAPL approach, that are useful in the more general and practical context that we target in this section.

Our approach to ARMAB is to start with a buffer-efficient, minimal valid schedule as in Section 6, but unlike the approach of Section 6, we iteratively try to encapsulate the minimal valid schedule within an outer loop having different iterations counts, which correspond to different candidates for the target blocking factor. In this context, the blocking factor can be modeled in the schedule tree as the loop iteration count of the root node: i.e., by setting $l(r) = J$, where $tree(r)$ is otherwise equivalent to the schedule tree of the original (minimal) schedule.

We have derived the following useful result characterizing the effects of "pushing" an loop iteration count into the leaf nodes of a schedule tree from the root of the tree.

Theorem 7 Suppose that we are given a schedule tree $T = tree(r)$ with $l(r) = 1$ and buffer cost B. Suppose that we are also given an integer $J > 1$, and that we derive the tree T' from T by simply setting $l(r) = J$. If in T', we then apply $\phi(r, V)$, where V is an integer factor of J, then we obtain $rate(T') = (A_0/V)$, where $A_0 = act(r)$ is the activations value for the original schedule tree T, and we also obtain a new buffer cost of $(V \times B)$.

Proof With $\phi(r, V)$ applied, we have an updated loop iteration count $l(r) = J/V$, and an updated activation count $act(r) \cdot (J/V)$ of T' (according to Theorem 2). Thus, the activation rate of the overall graph is $act(r)/V$. The new buffer cost is then $(V \times B)$ based on Theorem 3. □

Theorem 7 tells us that with tree root r as the FAPL target internal node, the activation rate and new buffer cost are independent of J beyond the requirement that they be derived from a FAPL factor V that divides J. This allows us to significantly prune the search space of candidate blocking factors.

Given an acyclic SDF graph, an initial SAS, and a buffer cost upper bound, the minimal activation rate can be computed through a finite number of steps. Suppose that all of the possible FAPL sequences $\Phi_1, \Phi_2, \ldots, \Phi_n$ are provided through exhaustive search of the given instance of the ARMUB problem. We denote as a_i and b_i the activation rate (under unit blocking factor) and buffer cost, respectively, that are derived through Φ_i $(1 \le i \le n)$.

Now suppose that we are given a user-specified buffer cost upper bound M. Then for each Φ_i, the maximum FAPL factor for the tree root r is $\lfloor M/b_i \rfloor$. Therefore, the minimal activation rate for the given ARMAB problem instance is $min_i(a_i/\lfloor M/b_i \rfloor)$.

However, this derivation is based on an exhaustive search approach that examines all possible FAPL sequences. For practical use on complex systems, more efficient methods are needed. In the next subsection, we develop an effective heuristic for this purpose.

7.1. *The FARM Algorithm*

Based on Theorem 7 and the developments of Section 6 we have developed an efficient heuristic to minimize the activation rate based on a given memory bound M. This heuristic, called *FARM (FAPL-based Activation-Rate Minimization)*, is outlined in Fig. 7. The algorithm iteratively examines the FAPL factors $V = 1, 2, \ldots, \lfloor M/B \rfloor$, where B is the buffer cost of the original schedule. The original schedule is, as with the technique of Section 6, derived from APGAN and GDPPO, which construct buffer-efficient minimal schedules (without any consideration of block processing). Here, $\lfloor x \rfloor$ repre-

```
Algorithm: FARM
Input: An SDF graph, $tree(r)$, and buffer cost upper bound $M$
Output: Minimum activation rate

let $B$ be the buffer cost induced from $tree(r)$.
set $MinRate = \infty$
for ($V = 1 \ldots \lfloor M/B \rfloor$) {
    copy $tree(r)$ to $tree'(r)$ with $l(r) = V$.
    execute $\phi(r, V)$ on $tree'(r)$
    run $GreedyFAPL$ on $tree'(r)$ to obtain minimum activations $act(r)$.
    if ($MinRate > (act(r)/V)$) {
        set $MinRate = act(r)/V$.
    }
}
return $MinRate$ as output
```

Figure 7. The FARM algorithm for the ARMAB (Activation Rate Minimization with Arbitrary Blocking factor) problem.

```
Algorithm: FARM-Sharing
Input: An SDF graph, $tree(r)$, and a buffer cost upper bound $M$
Output: Minimum activation rate

let $\delta$ be excess buffer cost ratio.
let $\epsilon$ be reduction rate.
set $MinRate = \infty$.
set excess buffer cost $\Delta = M \times \delta$ and $\Delta \in Z$.
while ($\Delta > 0$) {
    run FARM on the SDF graph, $tree(r)$, and buffer cost bound ($M + \Delta$).
    let $rate$ be the activation rate and $sched$ the SAS computed from FARM.
    let $\tau$ be the buffer sharing cost computed from any appropriate buffer
        sharing technique with the SDF graph and $sched$ as inputs.
    if ($\tau \leq M$ and $rate \leq MinRate$)
        $MinRate = rate$.
    else
        $\Delta = \Delta \times \epsilon$.
}
return $MinRate$ as output.
```

Figure 8. A FARM-based algorithm that integrates memory sharing techniques.

sents the largest integer that is less than or equal to the rational number x. For each of the candidate FAPL factors $V = 1, 2, \ldots, \lfloor M/B \rfloor$, the blocking factor is set to be equal to V, and the activation rate is minimized for that particular blocking factor.

The FARM algorithm keeps track of the minimal activation rate that is achieved over all iterations in which the memory constraint M is satisfied. The FAPL factor associated with this minimum activation rate and the resulting schedule tree configuration are returned as the output of the FARM algorithm.

The computational complexity of FARM is $O(\lfloor M/B \rfloor \Omega |V|(|V| + |E|))$.

7.2. Integration of FARM and Memory Sharing Techniques

Memory sharing is a useful technique to reduce the buffer cost requirement of an SDF application. In our ARMAB problem formulation, smaller activation rates are generally achievable if buffer sharing techniques are considered when specifying the buffer cost upper bound. In Fig. 8, we propose a FARM-based algorithm, called *FARM-Sharing*, that integrates buffer sharing techniques into the basic FARM framework.

The FARM-Sharing algorithm feeds larger buffer cost bounds to FARM initially, and gradually reduces the bounds as the algorithm progresses by employing buffer sharing techniques. The algorithm employs two special parameters δ and ϵ, which can be tuned by experimentation on various benchmarks. In our experiments, after tuning these parameters, we have consistently used $\delta = 10$, and $\epsilon = 0.5$.

Under buffer sharing, direct use of the buffer bound M will in general lead to buffer sharing implementations that overachieve the bound, and do not use the resulting slack to further reduce activation rate. Therefore, instead of directly using the user-specified bound of M, the FARM-sharing algorithm uses an internally increased bound. This increased bound is initially set to $M + \Delta$, where $\Delta = M\delta$. As the algorithm progresses, buffer sharing is used, and concurrently, the internally increased bound is gradually reduced towards a final value of M. Any suitable buffer sharing technique can be employed in this framework; in our experiments we have used the lifetime-based sharing techniques developed in [11]. In this way, significantly smaller activation rates can be generally achieved subject to the user-specified buffer bound M.

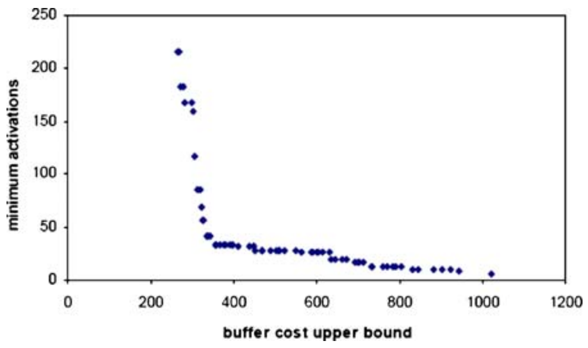


Figure 9. Trade-offs between activation rates and buffer costs of CD to DAT.

a

	16qam	4pam	aqmf1	aqmf2	aqmf3	aqmf4	cd2dat
DOO	100%	100%	99%	98%	100%	99%	97.1%

b

	16qam	4pam	aqmf1	aqmf2	aqmf3	aqmf4	cd2dat
DOO	100%	100%	99.8%	99.6%	100%	99.7%	98.8%

Figure 10. Effectiveness of **a** GreedyFAPL, **b** FARM, in terms of average DOO values.

8. Experiments

To demonstrate the trade-off between buffer cost minimization and activation rate optimization, exhaustive search is employed to the CD to DAT sample rate conversion example of Fig. 1. From the initial SAS of Fig. 1c, factor combinations of all loop iteration counts are exhaustively evaluated for the problem of ARMUB. The results are summarized in Fig. 9. Each dot in this chart is derived from a particular factor combination and the activation rates are obtained by $min(rate(S))$ (vertical axis) subject to $buf(S) \leq M$, where M is the buffer cost bound (horizontal axis).

Experiments are set up to compare the activation rates achieved by our heuristics to the optimum achievable activation rates that we determine by exhaustive search. Exhaustive search is used here to understand the performance of our heuristics; due to its high computational cost, such exhaustive search is not feasible for optimizing large-scale designs nor for extensive design space exploration even on moderate-scale designs.

The following DSP applications are examined in our experimental evaluation: *16qam* (a 16 QAM modem), *4pam* (a pulse amplitude modulation system), *aqmf* (filter bank), and *cd2dat* (CD to DAT sample rate conversion). These applications are ported from the library of SDF-based designs that are available in the Ptolemy design environment [3]. For each application, a number of buffer cost upper bounds (values of M) are selected uniformly in the range between the cost of the initial schedule (obtained from APGAN/GDPPO) to the cost of a flat schedule.

Given a buffer bound M, the *degree of optimality (DOO)* of GreedyFAPL or FARM is evaluated as $(rate_{opt}/rate_{sub}$, where $rate_{opt}$ is the optimal activation rate observed for ARMUB or ARMAB, and $rate_{sub}$ is the activation rate computed by GreedyFAPL or FARM. The degrees of optimality thus computed are averaged over the number of buffer bounds selected to obtain an average degree of optimality. The results are summarized in the table in Fig. 10 and they demonstrate the abilities of GreedyFAPL and FARM to achieve optimum solutions for ARMUB and ARMAB, respectively, most of the time.

To further evaluate the efficiency of our algorithms, randomly generated SDF graphs are experimented with. In these experiments, GreedyFAPL

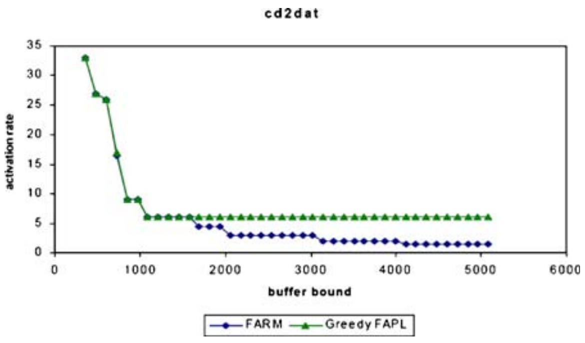


Figure 11. Comparison of GreedyFAPL and FARM.

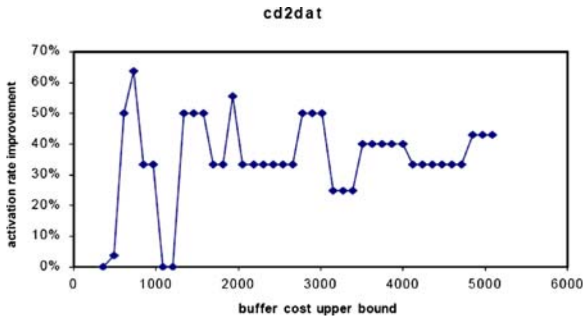


Figure 12. Activation rate improvement of FARM-Sharing over FARM.

a

	FIR	add	conv	DCT	cd2dat
fidelity	0.79	1	1	0	1

b

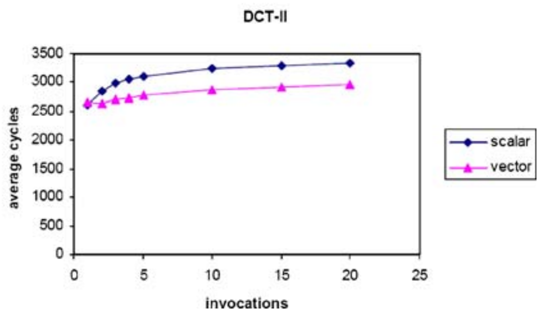


Figure 13. **a** Fidelity experiments. **b** Average latencies of DCT.

achieves optimal solutions approximately *90%* of the time with *98.35%* average DOO, and FARM achieves optimal solutions approximately *77%* of the time with *99%* average DOO.

To evaluate the impact of considering non-unity blocking factors, we performed experiments to compare GreedyFAPL and FARM for the CD to DAT example. In Fig. 11, both algorithms are investigated with several large buffer cost bounds. Due to the restriction of unit blocking factor, it is shown in the figure that, at a certain point, GreedyFAPL reaches a limit and cannot reduce the activation rate any further with increases in allowable buffer cost (loosening of the bound M) beyond that point. In contrast, FARM generally keeps reducing activation rates as long the buffer bound M is loosened.

The effectiveness of FARM-Sharing is explored as well in our experiments for the CD to DAT benchmark. The observed activation rate improvements achieved by FARM-Sharing over FARM are summarized in Fig. 12. Except for a few points, FARM-Sharing demonstrates over *20%* activation rate reduction. The actual degree of improvement achieved varies significantly across applications.

8.1. Suitability of the Activation Rate as a Performance Estimator

Some experiments are also conducted to evaluate the suitability of the activation rate as a high-level estimator for performance. In our context, the performance can be characterized as the average number of execution cycles required per graph iteration, which we refer to as the *average latency*. Let $L = (l_1, l_2, \ldots, l_n)$ denote the average latencies of an application under various blocking factor settings (these latencies are measured experimentally), and let $R = (r_1, r_2, \ldots, r_n)$ denote the corresponding activation rates (these can be calculated directly from the schedules). To measure the accuracy of activation-rate driven performance optimization, we use the estimation *fidelity*, as defined by

$$fidelity = \frac{2}{n(n-1)}\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} f_{ij}\right),$$

where $f_{ij} = 1$ if $sign(L_i - L_j) = sign(R_i - R_j)$, and $f_{ij} = 0$ otherwise.

Our fidelity experiments are summarized in Fig. 13 for four kernel functions (FIR, add, convolution, and DCT), and also for a complete application (CD to DAT conversion). Except for the DCT function, the activation rate is seen to be a good high level model for comparing different design points in terms of average latency. Average latencies of block processing implementations of the DCT increase with increasing vectorization degrees, however, the vectorized forms still demonstrate better performance compared to the scalar implementation (Fig. 13).

9. Conclusion

In this paper, we have first demonstrated the advantages of block processing implementation of DSP kernel functions. Then we have examined the integrated optimization problem of block processing, code size minimization, and data space reduction. We have shown that this problem can be modeled through a nonlinear programming formulation. However, due to the intractability of nonlinear programming, we have developed two efficient heuristics that are computationally efficient. We have evaluated both the GreedyFAPL and FARM algorithms, and our results demonstrate that they consistently derive high quality results. This paper has presented a number of concrete examples and additional bodies of experimental results that provide further insight into the relationships among block processing, memory requirements, and performance optimization for DSP software.

References

1. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Software Synthesis from Dataflow Graphs,” Kluwer, 1996.
2. R. E. Blahut, “Fast Algorithms for Digital Signal Processing,” Addison-Wesley, 1985.
3. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity—the Ptolemy approach,” *Proc. IEEE Spec. Issue Model. Des. Embed. Softw.*, vol. 91, no. 1, January 2003, pp. 127–144.
4. L. Guerra, M. Potkonjak, and J. Rabey, “System-level Design Guidance Using Algorithm Properties,” *IEEE Workshop VLSI Signal Proc.*, 1994, pp. 73–82.
5. I. Hong, M. Potkonjak, and M. Papaefthymiou, “Efficient Block Scheduling to Minimize Context Switching Time for Programmable Embedded Processors,” *Des. Autom. Embed. Syst.*, vol. 4, no. 4, 1999, pp. 311–327.
6. C. Hsu, M. Ko, and S. S., “Bhattacharyya, Software Synthesis from the Dataflow Interchange Format,” in *Proc. Int. Workshop Softw. Compilers Embed. Syst.*, Dallas, Texas, September 2005, pp. 37–49.

7. M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained Block Processing Optimization for Synthesis of DSP Software," in *Intl. Conf. Embed. Comput. Syst.: Archit., Model. Simul. (IC-SAMOS)*, Samos, Greece, July 2006, pp. 137–143.
8. K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak, "Optimizing Computations for Effective Block-processing," *ACM Transact. Des. Automat. Electron. Syst (TODAES)*, vol. 5, no. 3, July 2000, pp. 604–630.
9. E. A. Lee and D. G. Messerschmitt, "Synchronous Dataflow," *Proc. IEEE*, vol. 75, September 1987, pp. 1235–1245.
10. R. Leupers, "*Code Optimization Techniques for Embedded Processors—Methods, Algorithms, Tools*," Kluwer, 2000.
11. P. K. Murthy and S. S. Bhattacharyya, "Shared Buffer Implementations of Signal Processing Systems using Lifetime Analysis Techniques," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 2, February 2001, pp. 177–198.
12. K. K. Parhi, "VLSI Digital Signal Processing Systems: Design and Implementation," Wiley-Interscience, 1999.
13. S. Ritz, M. Willems, and H. Meyer, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," *Int. Conf. Acoust. Speech Signal Process (ICASSP)*, vol. 4, May 1995, pp. 2651–2654.
14. S. Ritz, M. Pankert, and H. Meyer, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," *Int. Conf. Appl. Spec. Array Process. (ASAP)*, October 1993, pp. 285–296.
15. W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of Hardware Software Codesign Workflow in PeaCE," in *Proc. Int. Conf. VLSI CAD*, October 1997.
16. W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. Int. Conf. Compiler Construction*, 2002.
17. V. Zivojnovic, S. Ritz, and H. Meyr, "Retimimg of DSP Programs for Optimum Vectorization," *Int. Conf. Acoust. Speech Signal Process (ICASSP)*, vol. 2, 1994, pp. 19–22.

Ming-Yung Ko received his B.S. degree in Computer Science from National Tsing-Hua University, Taiwan, R.O.C. in 1996 and the Ph.D. degree in Electrical and Computer Engineering from the University of Maryland, College Park in 2006. He is currently a Senior Engineer of Sandbridge Technologies, Inc., White Plains, NY, a system design firm of software-defined radio handsets. His research interest is in the area of embedded systems design with an emphasis on software synthesis, memory management, system software, and performance profiling.

Chung-Ching Shen received his B.S. degree in Computer and Information Science from National Chiao Tung University, Taiwan, R.O.C. in 2001 and the M.S. degree in Electrical and Computer Engineering from the University of Maryland, College Park in 2004. He is currently working towards the Ph.D. degree in Electrical and Computer Engineering from the University of Maryland, College Park. His research interests include energy-driven codesign for distributed embedded systems, signal processing, and low-power VLSI design for embedded systems in wireless sensor network applications.

Shuvra S. Bhattacharyya received his B.S. degree from the University of Wisconsin at Madison in 1987 and the Ph.D. degree from the University of California at Berkeley in 1994. He is a Professor in the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. He is also an Affiliate Associate Professor in the Department of Computer Science. He is coauthor of two books and the author or coauthor of more than 60 refereed technical articles. His research interests include signal processing, embedded software, and hardware/software codesign. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory, San Jose, CA, and as a Compiler Developer at Kuck & Associates, Champaign, IL.