Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, July 2006.

Memory-constrained Block Processing Optimization for Synthesis of DSP Software

Ming-Yung Ko, Chung-Ching Shen, and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies
University of Maryland, College Park MD 20742, USA

***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 take 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.

I. 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 [9]).

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 and Cocentric System Studio by Synopsys. Relevant research tools include DIF [6], Ptolemy [3], PeaCE [13], and StreamIt [14].

More specifically, in this paper, we examine the trade-off between block processing implementation and data memory requirements. Understanding this trade-off 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.

II. 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

This research was sponsored in part by the Semiconductor Research Corporation (contract number: 2001-HJ-905), and the Defense Advanced Research Projects Agency (through Management, Communications, and Control, Inc.).

[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. [12]. 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 [11]. Unlike the approach of [12], the work of [11] 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 [11] and [12], the optimization problem is formulated without user-specified data memory constraints. Furthermore, although, overall memory sharing cost is minimized in [11], 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 [7][15]. 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.

III. 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* 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*) [8] 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 [8], 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 an 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 $I_1, I_2, \ldots, I_m$ 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. Fig. 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.

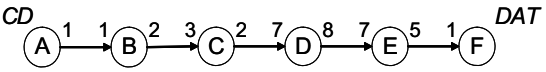
Any SAS can be transformed to an *R-schedule*, $S = (i_L S_L)(i_R S_R)$, where S_L (S_R) is the left (right) subschedule of S [1]. The binary structure of an R-schedule can be represented efficiently as a binary tree, which is called a *schedule tree* or just a *tree* in our discussion [10]. In this tree representation, every node is associated with an 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 $\lambda(tree(r))$.

(a)



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

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

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

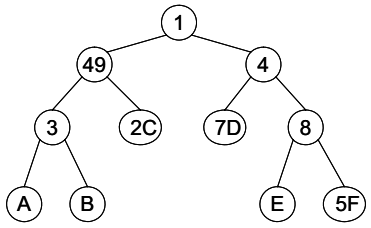


Fig. 2. A schedule tree example corresponding to the SAS of Fig. 1(c).

IV. 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 [12]. 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. 3(b) has less procedure call overhead, fast addressing through auto-increment modes, and better locality for pipelined execution compared to *add_scalar()* in Fig. 3(a).

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

(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);
}
```

Fig. 3. Integer addition (a) scalar version (b) vector version.

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.

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

V. 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, $act(r) = l(r)(act(left(r)) + act(right(r)))$.

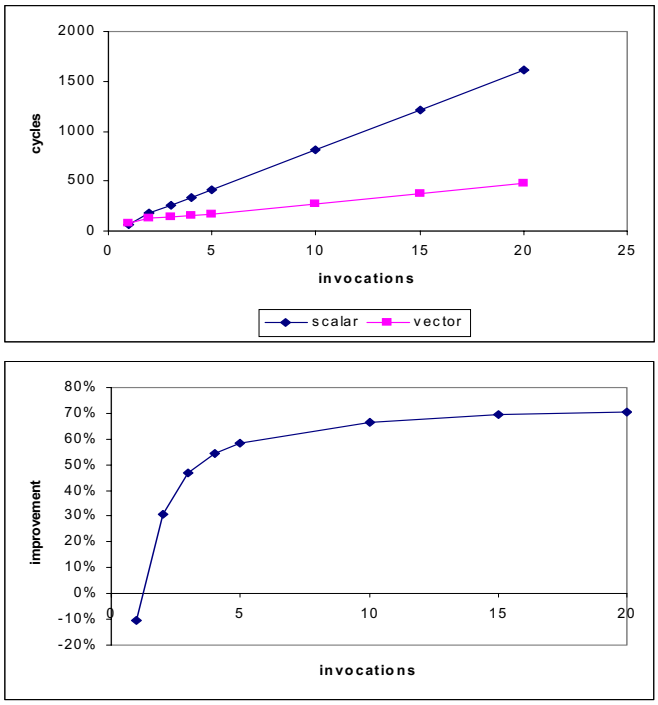


Fig. 4. Performance comparison of vectorized and scalar implementation of convolution operation.

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 of an SDF graph S, 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 III. 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.

VI. Minimization of Activation Rate

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).

One restriction in these formulations is that they assume that the input SDF graph is acyclic. 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.

In this paper, we investigate the problem of activation rate minimization under the constraint that the schedule blocking factor is unity. 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]. However, our developments on the activation rate minimization problem can easily be extended to handle arbitrary blocking factors. The details are omitted from this paper due to space restrictions.

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 counts of tree nodes in the schedule tree for S. The optimization variables are the set $\{l(x)\}$ of loop counts in the leaf nodes of the rearranged schedule tree (recall that these loop counts are equivalent to the vectorization degrees of the associated actors). The objective is to minimize the number of activations:

$$min(act(r)). \tag{1}$$

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

$$q(A) = \prod_{x \in path(a, r)} l(x) \quad \text{for each SDF actor } A, \tag{2}$$

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 counts are changed along the path, their product has to match the repetitions count of the associated actor.

In the activations minimization 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,

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

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

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

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

In summary, the activations minimization problem can be set up by casting Equations (1) through (4) into a *non-linear programming (NLP)* formulation, where the objective is given by (1), the variables are the loop iteration counts of the schedule tree nodes, and the constraints are given in (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 counts of all internal nodes to one, while setting the loop 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 and GDPPO algorithms are employed in this work to compute buffer-efficient SASs as a starting point for our memory-constrained block processing optimization [1].

A. Loop Count Factor Propagation

As described earlier, activations values of leaf nodes are always equal to one and independent of their loop counts. Hence, one approach to optimizing activations values is to enlarge the loop counts of leaf nodes by absorbing the loop counts of internal nodes. A similar approach is proposed in [12] to deal with cyclic SDF graphs with delays. The strategy in [12] 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 II, the work of [12] does not consider buffer cost in the optimization process.

For acyclic SDF graphs, as we will discuss later, factors of loop 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 count of r is $l(r)/V$, and for every leaf f in $tree(r)$ the new loop count is $V \cdot l(f)$. Loop 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 1: Given $tree(r)$ with $act(r)$, $\phi(r, V)$ reduces the activations of $tree(r)$ by a factor of 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 an *effective edge* from ϕ.

Theorem 2: 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.

Theorem 3: Given a valid schedule, the new schedule that results from a FAPL operation is also a valid schedule.

We omit the proofs of Theorems 1, 2, and 3 due to space restrictions.

B. 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 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 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. The computational complexity of GreedyFAPL is $O(\Omega|V|(|V|+|E|))$, where E is the edge set of the input SDF graph and Ω is the maximum number of factors in the prime factorization of the loop count of an internal node.

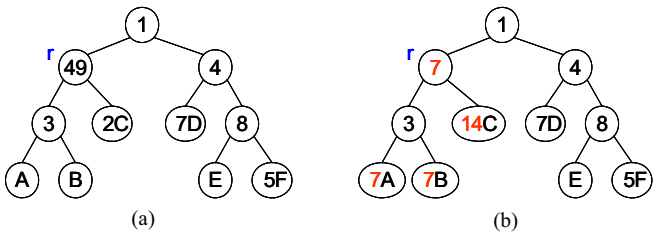


Fig. 5. With $\phi(r, 7)$, the tree in (a) turns to that in (b).

VII. 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 Figure 1. From the initial SAS of Fig. 1(c), factor combinations of all loop counts are exhaustively evaluated. The results are summarized in Fig. 7. Each dot in this chart is derived from a particular factor combination and the activations are obtained by $min(act(S))$ (vertical axis) subject to $buf(S) \le M$, where M is the buffer cost bound (horizontal axis).

The GreedyFAPL heuristic demonstrates high effectiveness in generating optimal solutions most of the time for activations minimization, or equivalently for activation rate minimization with unity blocking factor. Experiments are set up to compare the activation rates achieved by our heuristic to the optimum achievable activation rates that we determine by exhaustive search. Exhaustive search is used here to understand the performance of GreedyFAPL; 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.

```
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 π of l(a)) {
        compute overall buffer cost, C, as if
            φ(a, π) was run.
        if (C ≤ M) {
            execute φ(a, π).
        }
    }
}
return act(r) as output.
```

Fig. 6. The GreedyFAPL algorithm for activation rate minimization.

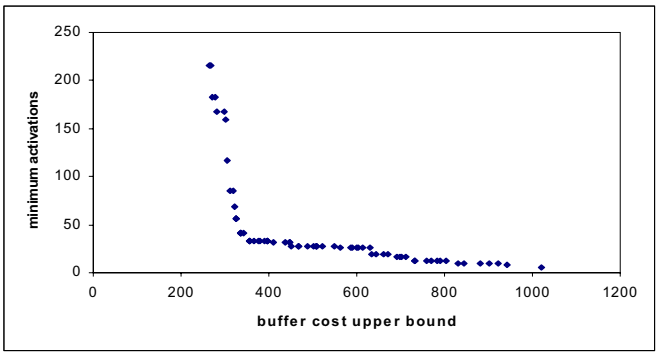


Fig. 7. Trade-off between activations and buffer costs of CD to DAT.

The following DSP applications are examined in our experimental evaluation: *16qam* (a 16 QAM modem), *4pam* (a pulse amplitude modulation system), *aqmf* (filterbank), 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 suboptimality (DOS)* of GreedyFAPL is evaluated as $(act_{sub} - act_{opt})/act_{opt}$, where act_{opt} is the optimal number of activations observed and act_{sub} is the number of activations computed by GreedyFAPL. The degrees of suboptimality thus computed are averaged over the number of buffer bounds selected to obtain an average degree of suboptimality. The results are summarized in Fig. 8, and they demonstrate the ability of GreedyFAPL to achieve optimum solutions most of the time.

To further evaluate the efficiency of our algorithms, randomly-generated SDF graphs are experimented with. In these experiments, GreedyFAPL achieves optimal solutions approximately 90% of the time.

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 $L = (l_1, \ldots, l_n)$ denote the average latencies of an application under various blocking factor settings (these latencies are measured experimentally), and let $R = (r_1, \ldots, r_n)$ 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. 9 for three kernel functions (FIR, add, and convolution), and also for a complete application (CD to DAT conversion). The activation rate is seen to be a good high level model for comparing different design points in terms of average latency.

	16qam	4pam	aqmf1	aqmf2	aqmf3	aqmf4	cd2dat
DOS	0%	0%	1%	2%	0%	1%	3%

Fig. 8. Effectiveness of GreedyFAPL for a number of applications.

	FIR	add	conv	cd2dat
fidelity	0.79	1	1	1

Fig. 9. Summary of fidelity experiments.

VIII. 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 a efficient heuristic that are computationally efficient. We have evaluated the GreedyFAPL heuristic, and our results demonstrate that it consistently derives 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 Academic Publishers. 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. *Proceedings of the IEEE*, January 2003.

[4] L.Guerra, M.Potkonjak, J.Rabey, "System-level design guidance using algorithm properties," *IEEE workshop VLSI in signal processing*, 73-82, 1994

[5] I. Hong, M. Potkonjak, and M. Papaefthymiou, "Efficient block scheduling to minimize context switching time for programmable embedded processors," *Design Automation for Embedded Systems*, 4(4):311-327, Kluwer Academic Publishers, 1999.

[6] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. *SCOPES*, pp. 37-49, Dallas, Texas, September 2005.

[7] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak, "Optimizing computations for effective block-processing," *TODAES*, 5(3):604-630, July 2000.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of IEEE*, 75:1235–1245, September 1987.

[9] R. Leupers. *Code Optimization Techniques for Embedded Processors — Methods, Algorithms, Tools*. Kluwer Academic Publishers, 2000.

[10] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Trans. on CAD*, 20(2):177-198, February 2001.

[11] S. Ritz, M. Willems, and H. Meyer, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," *ICASSP*, 4:2651-2654, May 1995.

[12] S. Ritz, M. Pankert, and H. Meyer, "Optimum vectorization of scalable synchronous dataflow graphs," *ASAP*, pp. 285-296, October 1993.

[13] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of hardware software codesign workflow in PeaCE," *International Conference on VLSI and CAD*, October 1997.

[14] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," *International Conference on Compiler Construction*, 2002.

[15] V. Zivojnovic, S. Ritz, and H. Meyr, "Retimimg of DSP programs for optimum vectorization," *ICASSP*, 2:19-22, 1994.