SYSTEMATIC CONSOLIDATION OF INPUT AND OUTPUT BUFFERS IN SYNCHRONOUS DATAFLOW SPECIFICATIONS[1]

Praveen K. Murthy
Angeles Design Systems
San Jose, CA

Shuvra S. Bhattacharyya
University of Maryland
College Park, MD

Abstract - Synchronous Dataflow, a subset of dataflow, is a commonly used model of computation in block diagram DSP programming environments. Because of the limited amount of memory in embedded DSPs, a key problem during software synthesis from SDF specifications is the minimization of the memory used by the target code. We develop a powerful formal technique called buffer merging that attempts to overlay buffers in the SDF graph systematically in order to drastically reduce data buffering requirements. This technique is complementary to lifetime-analysis based approaches, and we show that it can be fruitfully combined to yield a hybrid algorithm that results in less memory usage than either technique used alone. We give polynomial-time algorithms based on this formalism, and show that code synthesized using this technique results in a 45% reduction, on average, of the buffering memory consumption compared to existing techniques.

INTRODUCTION

Block diagram environments for DSPs have proliferated recently, with industrial tools like DSPCanvas from Angeles Design Systems, and COSSAP [13] from Synopsys, and academic tools like Ptolemy [4] from UC Berkeley, and GRAPE [6] from K. U. Leuven. Reasons for their popularity include ease-of-use, intuitive semantics, modularity, and strong formal properties of the underlying dataflow models.

Memory is an important metric for generating efficient code for DSPs used in embedded applications since most DSPs have very limited amounts of on-chip memory, and adding off-chip memory is frequently not a viable option due to the speed, power, and cost penalty this entails. High-level language compilers, like C compilers have been ineffective for generating good DSP code [17]; this is why most DSPs are still programmed manually in assembly language. However, this is a tedious, error-prone task at best, and the increasing complexity of the systems being implemented, with shorter design cycles, will require design development from a higher level of abstraction.

Most block diagram environments for DSPs that allow software synthesis, use

1. A portion of this research was sponsored by the US National Science Foundation (CAREER, MIP9734275).

the technique of threading for constructing software implementations. In this method, the block diagram is scheduled first. Then the code-generator steps through the schedule, and pieces together code for each actor that appears in the schedule by taking it from a predefined library. The code generator also performs memory allocation, and expands the macros for memory references in the generated code.

Clearly, the quality of the code will be heavily dependent on the schedule used. Hence, we consider in this paper scheduling strategies for minimizing memory usage. Since the scheduling techniques we develop operate on the coarse-grain, system level description, these techniques are somewhat orthogonal to the optimizations that might be employed by tools lower in the flow. For example, a general purpose compiler usually cannot make use of the global control and dataflow that our scheduler can exploit. Thus, the techniques we develop in this paper are complementary to the work being done on developing better procedural language compilers for DSPs [8][9]. Since the individual actors are programmed in procedural languages like 'C', the output of our SDF compiler is sent to a procedural language compiler to optimize the internals of each actor, and to possibly further optimize the code at a global level (for example, by performing global register allocation.)

The specific problem addressed by this paper is the following. Given a schedule for an SDF graph, there are several strategies that can be used for implementing the buffers needed on the edges of the graph. Previous work on minimizing these buffers has used two models: implementing each buffer separately [1][3][15], or using lifetime analysis techniques for sharing buffers [5][11][14]. In this paper, we present a third strategy—buffer merging. This strategy allows sharing of input and output buffers systematically, something that the lifetime-based approaches of [5][11][14] are unable to do. The reason that lifetime-based approaches break down when input/output edges are considered is because they make the conservative assumption that an output buffer becomes live as soon as an actor begins firing, and that an input buffer does not die until the actor has finished execution. Hence, the lifetimes of the input and output buffers overlap, and they cannot be shared. However, as we will show in this paper, this assumption can be relaxed if we look at the production and consumption pattern of individual tokens, and significant reuse opportunities exist as a result. However, the merging approach of this paper is in some sense, a dual of lifetime-based approaches because the merging technique is not able to exploit global sharing opportunities based on the topology of the graph and the schedule. It can only exploit sharing opportunities at the input/output level. Thus, we give a hybrid algorithm that combines both of these techniques and show that dramatic reductions in memory usage is possible compared to either technique used by itself.

In-place memory management strategies using array index instances are used in the Cathedral environment [16]; these strategies are applied to nested loop constructs in Silage. The merging approach presented in this paper is different from the approach of [16] in that it is specifically targeted to the high regularity and modularity present in single appearance schedule implementations (at the expense of decreased generality). In particular, the overlapping of SDF input/output buffers by shifting actor read and write pointers does not emerge in any straightforward way

from the more general techniques developed in [16]. Our form of buffer merging is especially well-suited for incorporation with the SDF vectorization techniques (for minimizing context-switch overhead) developed at the Aachen University of Technology [13] since the absence of nested loops in the vectorized schedules allows for more flexible merging of input/output buffers.

Ritz et al. [14] give an enumerative method for reducing buffer memory in SDF graphs; however, memory is a tertiary concern in their work and their techniques are not competitive with techniques that optimize for memory as a primary goal [3].

In [15], Sung et al. explore an optimization technique that combines procedure calls with inline code for single appearance schedules; this is beneficial whenever the graph has many different instantiations of the same basic actor. Thus, using parametrized procedure calls enables efficient code sharing and reduces code size even further. All of the scheduling techniques mentioned in this paper can use this code-sharing technique also, and our work is complementary to this optimization.

Dataflow is a natural model of computation to use as the underlying model for a block-diagram language for designing DSP systems. The blocks in the language correspond to actors in a dataflow graph, and the connections correspond to directed edges between the actors. These edges not only represent communication channels, conceptually implemented as FIFO queues, but also establish precedence constraints. An actor fires in a dataflow graph by removing tokens from its input edges and producing tokens on its output edges. The stream of tokens produced this way corresponds naturally to a discrete time signal in a DSP system. In this paper, we consider a subset of dataflow called synchronous dataflow (SDF) [7]. In SDF, each actor produces and consumes a fixed number of tokens, and these numbers are known at compile time. In addition, each edge has a fixed initial number of tokens, called delays.

NOTATION AND BACKGROUND

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor. Given an SDF edge e, we denote the source actor, sink actor, and delay (initial tokens) of e by $src(e)$, $snk(e)$, and $del(e)$. Also, $prd(e)$ and $cns(e)$ denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$. If $prd(e) = cns(e) = 1$ for all edges e, the graph is called **homogenous**. In general, each edge has a FIFO buffer; the number of tokens in this buffer defines the state of the edge. Initial tokens on an edge are just initial tokens in the buffer. The size of this buffer can be determined at compile time, as shown below. The state of the graph is defined by the states of all edges.

A **schedule** is a sequence of actor firings. We compile an SDF graph by first constructing a **valid schedule** — a finite schedule that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge (i.e, returns the graph to its initial state). We represent the minimum number of times each actor must be fired in a valid schedule by a vector q_G, indexed by the actors in G (we often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the **balance equations** for G, which specify that q must

satisfy $prd(e)q(src(e)) = cns(e)q(snk(e))$, for all edges e in G. The vector q, when it exists, is called the **repetitions vector** of G, and can be computed efficiently [3].

CONSTRUCTING MEMORY-EFFICIENT LOOP STRUCTURES

In [3], the concept and motivation behind **single appearance schedules** (**SAS**) has been defined and shown to yield an optimally compact inline implementation of an SDF graph with regard to code size (neglecting the code size overhead associated with the loop control). An SAS is a schedule in which each actor appears only once when loop notation is used. Figure 1 shows an SDF graph, and valid schedules for it. The notation $2B$ represents the firing sequence BB. Similarly, $2(B(2C))$ represents the schedule loop with firing sequence $BCCBCC$. Schedules 2 and 3 in figure 1 are single appearance schedules since actors A, B, C appear only once. An SAS like the third one in Figure 1(b) is called **flat** since it does not have any nested loops. In general, there can be exponentially many ways of nesting loops in a flat SAS.

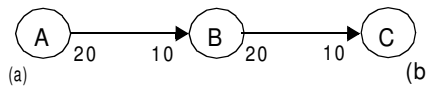
Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Figure 1(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 40, 60, and 50 respectively.

OPTIMIZING FOR BUFFER MEMORY

Following [3][10], we give priority to code-size minimization over buffer memory minimization. Hence, the problem we tackle is one of finding buffer-memory-optimal SAS, since this will give us the best schedule in terms of buffer-memory consumption amongst the schedules that have minimum code size. Following [3] and [10], we also concentrate on acyclic SDF graphs since algorithms for acyclic graphs can be used in the general SAS framework developed in [3].

For an acyclic SDF graph, any topological sort $a\ b\ c\ldots$ immediately leads to a valid flat SAS given by $(q(a)a)\ (q(b)b)\ldots$. Each such flat SAS leads to a set of SASs corresponding to different nesting orders.

In [10] and [3], the buffering cost was defined as the sum of the buffer sizes on each edge, assuming that each buffer is implemented separately, without any sharing. In this paper, we use an alternative cost for implementing buffers. Our cost is based on overlaying buffers so that spaces can be re-used when the data is no longer needed. This technique is called **buffer merging**, since, as we will show, merging an input buffer with an output buffer will result in significantly less space required than their sums.



Valid Schedules

(1): ABCBCCC (2): A(2 B(2 C))

(3): A(2 B)(4 C) (4): A(2 BC)(2 C)

Fig 1. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

MERGING AN INPUT/OUTPUT BUFFER PAIR

Consider the second schedule in figure 1(b). If each buffer is implemented separately for this schedule, the required buffers on edges AB and BC will be of sizes 20 and 20, giving a total requirement of 40. Suppose, however, that it is known that B consumes its 10 tokens per firing *before* it writes any of the 20 tokens. Then, when B fires for the first time, it will read 10 tokens from the buffer on AB, leaving 10 tokens there. Now it will write 20 tokens. At this point, there are 30 live tokens. If we continue observing the token traffic as this schedule evolves, it will be seen that 30 is the maximum number that are live at any given time. Hence, we see that in reality, we only need a buffer of size 30 to implement AB and BC. Indeed, the diagram shown in figure 2 shows how the read and write pointers for actor B would be overlaid, with the pointers moving right as tokens are read and written. As can be seen, the write pointer, X(w,BC) never overtakes the read pointer X(r,AB), and the size of 30 suffices. Hence, we have merged the input buffer (of size 20) with the output buffer (of size 20) by overlapping a certain amount that is not needed because of the lifetimes of the tokens.

In past work on dataflow, the execution of an actor is treated as atomic for most purposes. However, as the above example shows, it is useful to know when precisely tokens are produced and consumed during an actor's execution as this enables better memory management. This idea can be formalized using the **consumed-before-produced (CBP)** parameter [2].

The CBP parameter

We define a parameter called the **consume-before-produce (CBP)** value; this parameter is a property of the SDF actor and a particular input/output edge pair of that actor. Informally, it gives the best known lower bound on the difference between the number of tokens consumed and number of tokens produced over the entire time that the actor is in the process of firing. Formally, let X be an SDF actor, and let e_i be an input edge of X and e_o an output edge of X. Let the firing of X begin at time 0 and end at time T. Define $c(t)$ $(p(t))$ to be the number of tokens that have been consumed (produced) from (on) e_i (e_o) by time $t \in [0, T]$. Then, we define

$$CBP(X, e_i, e_o) = MIN_t\{c(t) - p(t)\} \quad (1)$$

We can then formally state the size of the buffer that results when input/output buffers are merged using the CBP parameter and parameters computed from the SAS; these results may be found in [12]. Since stating these formulas requires a certain amount of loop scheduling machinery to be developed and described, we omit this here due to space considerations as it is not necessary for the main result

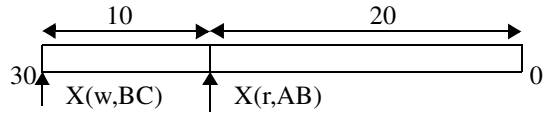


Fig 2. The merged buffer for implementing edges AB and BC in figure 1.

in this paper, which is to develop a merging algorithm for arbitrary, acyclic SDF graphs. It suffices to know that we can compute the size of the merged buffer for an input/output edge pair, given the CBP parameter and an SAS.

Consider a pair of input/output edges e_i, e_o for an actor Y. Let $X = src(e_i)$, $Z = snk(e_o)$ $snk(e_i) = Y = src(e_o)$. Let $b_i \oplus b_o$ denote the buffer resulting from merging the buffers b_i and b_o, on edges e_i and e_o respectively. Define $|b|$ to be the size of buffer b.

Lemma 1:[12] The size of the merged buffer is no greater than the sum of the buffer sizes implemented separately.

Define the augmentation function $A(b_i \oplus b_o)$ to be the amount by which the output buffer b_o has to be augmented due to the merge $b_i \oplus b_o$. That is, $A(b_i \oplus b_o) = |b_i \oplus b_o| - |b_o|$.

Theorem 1: The merge operator is associative; i.e, if $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$ is a chain of four actors, $e_i = (v_i, v_{i+1}), i = 1, 2, 3$, and b_i are the respective buffers, then

$$|(b_1 \oplus b_2) \oplus b_3| = |b_1 \oplus (b_2 \oplus b_3)|.$$

Theorem 2:[12] Let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ be a path (a chain of actors and edges) in the SDF graph. Let b_i be the buffer on the output edge of actor v_i, and let S be a given SAS (according to which the b_i are determined). Then,

$$|b_1 \oplus \ldots \oplus b_{k-1}| = \sum_{i=2}^{k-1} A(b_{i-1} \oplus b_i) + |b_{k-1}| \quad (2)$$

ACYCLIC GRAPHS

In this section, we develop the merging technique for arbitrary, delayless, acyclic SDF graphs. The techniques we develop here can easily be extended to handle graphs that have delays [12]. SASs for SDF graphs that contain cycles can be constructed in an efficient and general manner by using the loose interdependence scheduling framework of [3].

When acyclic graphs are considered, there are two dimensions that come into play for designing merging algorithms. The first dimension is the choice of the topological ordering of the actors; each topological ordering leads to a set of SASs. This dimension has been extensively dealt with before in [3], where two heuristic approaches were devised for determining good topological orderings. While these heuristics were optimized for minimizing the buffer memory cost function where each buffer is implemented separately, they can be used with the new merged cost function as well. We leave for future work to design better heuristics for the merged cost function, if it is possible.

The second dimension is unique to the merge cost function, and is the issue of the set of paths that buffers should be merged on. In other words, given a topological sort of the graph, and a nested SAS for this graph, there still remains the issue of what paths buffers should be merged on. Since an acyclic graph can have an exponential number of paths, it is necessary to have efficient techniques for determining

these paths. In the following section, we develop a bottom up approach that combines lifetime analysis techniques from [11] and the merging approach to generate implementations that arguably extract the maximum benefit of both approaches.

A bottom-up approach

We describe a technique for determining merge paths that also combines lifetime analysis techniques from [11]. Briefly, the lifetime analysis techniques developed in [11] construct an SAS optimized using a particular shared-buffer model that exploits temporal disjointedness of the buffer lifetimes. The method then constructs an intersection graph that models buffer lifetimes by nodes and edges between nodes if the lifetimes intersect in time. FirstFit allocation heuristics [11] are then used to perform memory allocation on the intersection graph. The shared buffer model used in [11] is useful for modeling the sharing opportunities that are present in the SDF graph as a whole, but is unable to model the sharing opportunities that are present at the input/output buffers of a single actor. The model has to make the conservative assumption that all input buffers are simultaneously live with all output buffers of an actor while the actor has not fired the requisite number of times in the periodic schedule. This means that input/output buffers of a single actor cannot be shared under this model. However, the buffer merging technique developed in this paper models the input/output edge case very well, and is able to exploit the maximum amount of sharing opportunities. However, the merging process is not well suited for exploiting the overall sharing opportunities present in the graph, as that is better modeled by lifetime analysis. Hence, the bottom-up approach we give here combines both these techniques, and allows maximum exploitation of sharing opportunities at both the global level of the overall graph, and the local level of an individual input/output buffer pair of an actor.

The algorithm is stated in figure 3. It makes several passes through the graph, each time merging a suitable pair of input/output buffers. For each merge, a global memory allocation is performed using the combined lifetime of the merged buffer. That is, the start time of the merged buffer is the start time of the input buffer, and the end time is the end time of the output buffer (the procedure `chngIntsect-Graph` performs this). If the allocation improves, then the merge is recorded (procedure `recordMerge`). After examining each node and each pair of input/output edge pairs, we determine whether the best recorded merge improved the allocation. If it did, then the merge is performed (procedure `mergeRecorded`), and another pass is made through the graph where every node and its input/output edge pairs is examined. The algorithm stops when there is no further improvement.

Running time analysis

The loops labelled (1), (2), and (3) take

$$\sum_{i=1}^{|V|} indeg(v_i) \cdot outdeg(v_i) \tag{3}$$

steps, where $indeg(v)$ is the in-degree of actor v and $outdeg(v)$ is the out-degree of actor v. In the worst possible case, we can show that this sum is $O(|V|^3)$,

assuming a dense, acyclic graph. If the graph is not dense, and the in- and out-degrees of the actors are bounded by pre-defined constants, as they usually are in most SDF specifications, then equation 3 would be $O(|V|)$. The merging step in line (4) can be precomputed and stored in a matrix since merging a buffer with a chain of merged buffers just involves merging the buffer at the end of the chain and summing the augmentation. This precomputation would store the results in an $|E| \times |E|$ matrix, and would take time $O(|E|^2 \cdot \log|V|)$ in the average case, and $O(|E|^2 \cdot |V|)$ time in the worst case. So line (4) would end up taking a constant amount of time since the precomputation would occur before the loops. The intersectionGraph procedure can take $O(|E|^2)$ time in the worst case. While this could be improved by recognizing the incremental change that actually occurs to the lifetimes, it is still hampered by the fact that the actual allocation heuristic still takes time $O(|E|^2)$. The overall while loop can take $O(|E|)$ steps since each edge could end up being merged. Hence, the overall running time, for practical systems, is $O(|V| \cdot |E|^3)$ which is $O(|V|^4)$ for sparse graphs. Improvement, if any, can be achieved by exploring ways of implementing the FirstFit heuristic to work incrementally (so that it does not take $O(|E|^2)$); however, this is unlikely to be possible

```
Procedure mergeBottomUp(SDF Graph G, SAS S)

I ← computeIntersectionGraph(G, S)
cur_best ← allocate(I)
iter_best ← cur_best

while (true)
    for each node v ∈ G                    (1)
        for each input edge e_i of v       (2)
            for each output edge e_o of v (3)
                b ← b_i ⊕ b_o              (4)
                I ← chngIntsectGraph(S, b)
                m ← allocate(I)
                if (m < iter_best)
                    recordMerge(b, I, m)
                    iter_best ← m
                fi
                restore(I, b_i, b_o, S)
            end for
        end for
    end for
    if (iter_best < cur_best)
        cur_best ← iter_best
        mergeRecorded()
    else
        break
```

Fig 3. A bottom-up approach that combines buffer merging with lifetime analysis.

as the merged buffer will have a different lifetime and size, and the allocation has to be redone from scratch each time.

EXPERIMENTAL RESULTS

We have tested this algorithm on several practical benchmark examples; table 3

Table 3: Performance on practical systems

System	non-shared	shared	shared and merged	% Imp
q23	1271	492	245	50.2
q12	342	58	30	48.3
q235	492	240	110	54.2
q235_5d	8967	5690	3790	33.4
satrec	1542	991	720	27.3
FFT	1222	514	258	49.8
phArr	2496	2071	1234	40.4

shows these results. The first four systems are quadrature mirror filterbank systems used for audio and image coding. The fourth example is a satellite receiver from [14]. The last two examples are a an overlapped-add FFT implementation, and a phased array system for detecting signals. The second column gives the best cost obtainable when no buffer sharing is used, and each buffer is implemented separately. The third column gives the best shared implementation using the techniques in [11]. The fourth column shows the results obtained using the bottom-up algorithm. As can be seen, the improvement of the techniques in this paper over previous work is on average around 45% (last column), and as high as 54%.

CONCLUSION

We have developed a novel algorithm for applying the buffer merging technique in acyclic SDF graphs. The algorithm we have given is a hybrid algorithm that combines lifetime analysis techniques with the merging technique to determine an optimal set of merges. We have shown that the benefit of using buffer merging can result in up to a 50% improvement over previous methods buffer optimization.

Earlier work on SDF buffer optimization has focused on the separate buffer model, and the lifetime model, in which buffers cannot share memory space if they simultaneously contain live data. Our work on buffer merging in this paper has formally introduced a third model of buffer implementation in which input and output buffers can be overlaid in memory even if their lifetimes overlap. We have shown that our merging techniques can produce large improvements over the separate-buffer and lifetime-based implementation. However, buffer merging does not render separate-buffers or lifetime-based buffer sharing obsolete. Separate buffers are useful for implementing edges that contain delays efficiently. Furthermore, they

provide a tractable cost function with which once can rigorously prove useful results on upper bound memory requirements [3]. Lifetime-based sharing is a dual of the merging approach, as mentioned already, and can be fruitfully combined with the merging technique to develop a powerful hybrid approach that is better than either technique used alone, as we have demonstrated with the bottom-up algorithm.

REFERENCES

[1] M. Ade, R. Lauwereins, J. Peperstraete, "Implementing DSP applications on heterogeneous targets using minimal size data buffers," IEEE Wkshp. on Rapid System Prototyping, 1996.

[2] S. S. Bhattacharyya, P. K. Murthy, "The CBP parameter — a useful annotation to aid block diagram compilers for DSP," Proc. of ISCAS, Geneva, Switzerland, May 2000.

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

[4] J. Buck, S. Ha, E. A. Lee, D. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems," *Intl. J. of Computer Simulation*, Jan. 1995.

[5] E. De Greef, F. Catthoor, H. De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems," Intl. Conf. on ASAP, 1997.

[6] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, "Geometric parallelism and cyclo-static data flow in GRAPE-II," IEEE Wkshp. Rapid System Prototyping, 1994.

[7] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, Feb., 1987.

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Code optimization techniques in embedded DSP microprocessors," *DAES*, Jan. 1998.

[9] P. Marwedel, G. Goossens, eds, *Code generation for embedded processors*, Kluwer, 1995.

[10] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, "Joint Code and Data Minimization for Synchronous Dataflow Graphs," *J. Formal Methods in System Design*, July 1997.

[11] P. K. Murthy, S. S. Bhattacharyya, "Shared Memory Implementations of Synchronous Dataflow Specifications Using Lifetime Analysis Techniques," Tech. rept. UMIACS-TR-99-32, www.cs.umd.edu/TRs/TRumiacs.html, June 1999.

[12] P. K. Murthy, S. S. Bhattacharyya, "Buffer Merging—A Powerful Technique for Reducing Memory Requirements of SDF Specifications," ISSS, 1999.

[13] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," Intl. Conf. on ASAP, 1993.

[14] S. Ritz, M. Willems, H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," ICASSP, 1995.

[15] W. Sung, J. Kim, S. Ha, "Memory efficient synthesis from dataflow graphs," ISSS, 1998.

[16] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man, "In-place memory management of algebraic algorithms on application specific ICs," *J. VLSI SP*, 1991.

[17] V. Zivojnovic, J. M. Velarde, C. Schlager, H. Meyr, "DSPStone — A DSP-oriented Benchmarking Methodology," ICSPAT, 1994.