Memory Size Estimation for Multimedia Applications*

Peter Grun, UC Irvine, pgrun@ics.uci.edu
Florin Balasa, Rockwell Intn'l, balasaf@rss.rockwell.com
Nikil Dutt, UC Irvine, dutt@ics.uci.edu

Abstract

Memory modules dominate the cost, performance, and power of embedded systems that process multidimensional signals, typically present in image and video processing. Therefore, studying the impact of parallelism on memory size is crucial for trading off system performance against area cost, to enable intelligent system partitioning and exploration. We propose a memory size estimation method for algorithmic specifications containing multidimensional arrays and parallel constructs, intended as part of a high-level partitioning and exploration methodology. The system designer can trade-off estimation accuracy for increased run time. We present the results of our estimation approach on a number of image and video processing kernels, and discuss some preliminary results on the influence of parallelism on storage requirement.

1 Introduction

In the design of embedded HW/SW systems, the high level design space exploration step is critical for obtaining a cost effective implementation. Decisions at this level have the highest impact on the final result. The step of partitioning the initial specification into HW/SW, as well as between different processing units, together with decisions regarding the parallelism inherent in such designs, allow trading-off the system performance against cost. To drive this process, a set of fast estimation tools is crucial.

Varying the code parallelism by means of code transformations [2] (e.g., instruction reordering, loop splitting, fusion, fission, interchanging, skewing [14]), or by assigning code portions to different partitions, may result in significant performance variations for the system implementation. More instructions executed in parallel lead usually to higher performance, but hardware cost may grow substantially.

In the algorithmic specifications of image and video applications the multidimensional variables (signals) are the main data structures. These large arrays of signals have to be stored in on-chip and off-chip memories. In such applications, memory often proves to be the most important hardware resource. Therefore it is important to be able to predict after a set of (parallelizing) transformations and partitioning steps the memory requirements for every design alternative.

We propose a method for memory size estimation, targeting specifications with multidimensional arrays, containing both instruction level (fine-grain) and coarse-grain parallelism [2], [14]. The impact of parallelism on memory size has not been previously studied in a consistent way. Together with tools for estimating the area of functional units and the performance of the design, our memory estimation approach can be used in a high level exploration methodology to trade-off performance against system cost.

Our paper is organized as follows. Section 2 briefly reviews some major results obtained in the field of memory estimation. Section 3 defines the memory size estimation problem. Our approach is presented in Section 4. In Section 5 we discuss the influence of parallelism on memory size. Our experimental results are summarized in Section 6, followed by the conclusions and our future directions of research in Section 7.

2 Previous work

One of the earliest approaches to memory estimation is the left edge algorithm [5], which assigns the scalar variables to registers. This approach is not suited for multidimensional signal processing applications, due to the prohibitive computational effort.

One of the earliest approaches of handling arrays of signals is based on clustering the arrays into memory modules such that a cost function is minimized [9]. The possibility of signals with disjoint life times to share common storage locations is however ignored, the resulting memory requirements often significantly exceeding the actual storage needed. More recently, [10] proposed a more refined array clustering, along with a technique for binding groups of arrays to memory modules drawn from a given library. However, it seems that the technique does not perform in-place mapping within an array.

Approaches which deal with large multidimensional arrays operate on non-procedural [1] and stream models [7]. Non-procedural specifications do not have enough information to estimate accurately the memory size, since by changing the instruction sequence, large memory variations are produced. For example, assuming the code in Figure 1a is non-procedural, the memory size could vary between 100 and 150 locations, as in Figure 1b. [13] uses a data-flow oriented view, as [1], but has good results for simpler specifications (constant loop bounds, simpler indices). [12] modified the loop hierarchy and the execution sequence of the source code, by placing polyhedrons of signals derived from the operands in a common space and determining an ordering vector in that space. None of the above techniques addresses parallel specifications.

* This research is partially supported by NSF Grant MIP-9708067.

1092-6100/98 $10.00 © 1998 IEEE

Other memory management approaches use the access frequencies to balance the port usage [11], and to optimize the partitioning between on-chip scratch-pad and cache-accessed memories [6]. They do not consider the memory size though.

3 The memory estimation problem

The problem of memory size estimation is, given an input algorithmic specification containing multidimensional arrays, predict what is the number of memory locations necessary to satisfy the storage requirements of the system.

The ability to predict the memory characteristics of behavioral specifications without synthesizing them is vital to producing high quality designs with reasonable turnaround. During the HW/SW partitioning and design space exploration phase the memory size varies considerably. For example, in Figure 1, by assigning the second and third loop to different HW/SW partitions, the memory requirement changes by 50% (we assume that array *a* is no longer needed and can be overwritten). Here the production of the array *b* increases the memory by 50, without consuming values. On the other hand, the loop producing array *c* consumes 100 values (2 per iteration). Thus, it is beneficial to produce the array *c* earlier, and reuse the memory space made available by array *a*. By producing *b* and *c* in parallel, the memory requirement is 100.

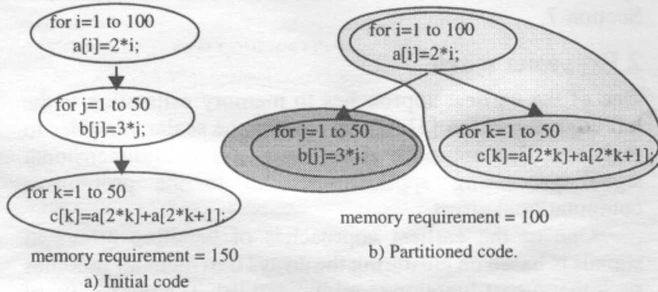

Figure 1. Memory size variation during partitioning/parallelization

Our estimation approach considers such reuse of space, and gives a fast estimate of the memory size. To allow High-Level design decisions, it is very important to provide good memory size estimates with reasonable computation effort, without having to perform complete memory assignment for each design alternative. During synthesis, when the memory assignment is done, it is necessary to make sure that different arrays (or parts of arrays) with non-overlapping lifetimes share the same space. The work in [3] addresses this problem, obtaining results close to optimal. Of course, by increasing sharing between different arrays, the addressing becomes more complex, but in the case of large arrays, it is worth increasing the cost of the addressing unit in order to reduce the memory size.

Our memory size estimation approach uses elements of the polyhedral data-flow analysis model introduced in [1], with the following major differences:

(1) The input specifications may contain explicit constructs for parallel execution. This represents a significant extension required for design space exploration, and is not supported by any of the previous memory estimation /allocation approaches mentioned in Section 2.

(2) The input specifications are interpreted procedurally, thus considering the operation ordering consistent with the source code. Most of the previous approaches operated on non-procedural specifications, but in practice a large segment of embedded applications market (e.g., DSPStone benchmark suite [15]) operates on procedural descriptions, so we consider it is necessary to accommodate also these methodologies.

Our memory size estimation tool handles specifications containing nested loops having affine boundaries. The operands can be multidimensional signals with (complex) affine indices. The parallelism is explicitly described by means of *cobegin-coend* and *forall* constructs. This parallelism could be described explicitly by the user in the input specification, or could be generated through parallelizing transformations on procedural code. We assume the input has the single-assignment property (this could be generated through a preprocessing step).

The output of our memory estimation tool is a range of the memory size, defined by a lower- and upper-bound. The predicted memory size for the input algorithm lies within this range, and in most of the cases it is close to the lower-bound (see the experiments). Thus, we see the lower bound as a prediction of the expected memory size, while the upper bound gives an idea of the accuracy of the prediction (i.e., the error margin). When the two bounds are equal, an "exact" memory size evaluation is achieved (by exact we mean the best that can be achieved with the information available at this step, without doing the actual memory assignment). In order to handle complex specifications, we provide a mechanism to trade-off the accuracy of predicting the storage range against the computational effort.

4 Memory estimation approach

Our memory estimation approach (called MemoRex) has two parts. Starting from a high level description which may contain also parallel constructs, the Memory Behavior Analysis (MBA) phase analyzes the memory size variation, by approximating the memory trace, as shown in Figure 2, using a covering bounding area. Then, the Memory Size Prediction (MSP) computes the memory size range, which is the output of the estimator. The backward dotted arrow in Figure 2 shows that the accuracy can be increased by subsequent passes.

The memory trace represents the size of the occupied storage in each logical time step during the execution of the input algorithm. When dealing with complex specifications, we do not determine the exact memory trace (the continuous line in the graphic from Figure 2) due to the high computational effort required. A bounding area encompassing the memory trace - the shaded rectangles from the graphic in Figure 2 - is determined instead.

The storage requirement of an input specification is obviously the peak of the (continuous) trace. When the memory trace cannot be determined exactly, the approximating bounding area can provide the lower- and upper- bounds of the trace peak. This range of the memory requirement represents the result of our estimation tool.

The MemoRex algorithm (Figure 3) has five steps.

Employing the terminology introduced in [12], the first step computes the number of array elements produced by each definition domain and consumed by each operand domain. The definition/operand domains are the array references in the left/right hand side of the assignments.

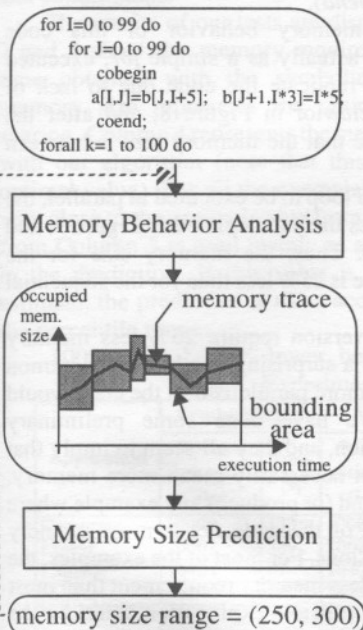

Figure 2. Flow of the MemoRex algorithm

Memory Behavior Analysis
1. Perform data-dependence analysis
2. Compute memory size between loop nests
3. Determine bounding area for the memory trace

Memory Size Prediction
4. Determine memory size range

Accuracy increase
5. Ifmore accuracy needed, split the critical rectangles and goto step 1

Figure 3. The MemoRex algorithm

```
for I=1 to 10 do
    a[I]=I;
for I=1 to 10 do
    b[I]=a[I]+a[11-I];
for I=1 to 5 do
    c[I]=a[2*I]+b[2*I]+
            b[2*I+1];
```

Figure 4. Illustrative example

Step 2 determines the occupied memory size at the boundaries between the loop nests. For instance, such an imaginary boundary succeeds the two nested loops and precedes the *forall* loop in the code from Figure 2. In fact, Step 2 determines a set of points *on the memory trace*. To determine or approximate the unknown parts of the trace, Step 3 determines a set of covering bounding rectangles, represented as shaded rectangles in Figure 2. This is the output of the Memory Behavior Analysis part of our algorithm. Based on the memory behavior, Step 4 approximates the trace peak, determining the range for the memory requirement.

Step 5 refines the bounding area of the memory trace by breaking up the larger rectangles into smaller ones. The resulting bounding area approximates more accurately the shape of the memory trace, and the resulting range for the memory requirement will get narrower.

In the following, we will employ for illustration the simple code in Figure 4.

4.1 Data-dependence analysis

By studying the dependence relations between the array references in the code, this step determines the number of array elements produced (born) or consumed (dying) during each assignment.

The number of array elements *produced* by an assignment is given by the size of the corresponding definition domains. In the illustrative example, the number of array elements produced in the three loops are $Card\{a[I], 1<=I<=10\} = 10$, $Card\{b[I], 1<=I<=10\} = 10$, and $Card\{c[I], 1<=I<=5\} = 5$, respectively. In general though, the size of signal domains is more difficult to compute, for instance, when handling array references within the scope of loop nests and conditions: our tool employs the algorithm which determines the size of linearly bounded lattices, described in [1].

On the other hand, the number of array elements consumed by an operand domain, is not always equal to the size of the operand domain, as some of the array elements may belong also to other operands from subsequent assignments. For instance, only two array elements are consumed by the operand domain *a[I]* (i.e., *a[7], a[9]*) and three other array elements (*a[1], a[3], a[5]*) by the operand domain *a[11-I]*, as the other *a* array elements of even index are read also by the operand *a[2*I]* in the third loop.

In general, the computation of dying signals is more complicated when dealing with loop nests and conditions. We perform the computation employing a symbolic time function for each assignment, which characterizes (in a closed form formula) when each array element is read, thus allowing us to find the last read of each array element, i.e., the domain consuming that element.

4.2 Computing the memory size between loop nests

For each nest of loops, the total number of memory locations produced/consumed is the sum of the locations produced/consumed in each domain within that loop nest. The memory size after executing a loop nest is the sum of the memory size at the beginning of the loop nest, and the number of array elements produced minus the number of array elements consumed within the loop nest:

mem(end_loop) = mem(begin_loop) + sum(prod's) - sum(consmp's)

As shown in Figure 5a, the memory size for our example is 0 at the beginning of the first loop, 10 after the execution of the first loop (because this loop produces 10 array elements and does not consume any), and 15 at the end of the second loop, as 10 new array elements are produced (*b[1..10]*), while the five odd-index array elements *a* are no longer necessary.

4.3 Determining the bounding rectangles

Based on the inform3ation already acquired in Steps 1 and 2, our algorithm constructs bounding rectangles for each loop nest in the specification. These rectangles are built such that they cover the memory trace (see Figure 5b). Thus, they characterize the behavior of the memory size for the portion of code under analysis.

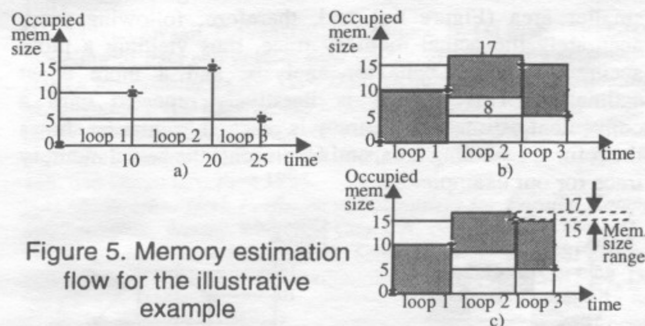

Figure 5. Memory estimation flow for the illustrative example

We illustrate the construction of the bounding rectangles for the second loop in Figure 4. It is known from Step 1 that 10 array elements (*b[1..10]*) are produced, while

the operand domain *a[I]* consumes 2 array elements, and the operand domain *a[11-I]* consumes 3. Since at most 7 out of the 10 assignments may not consume any values (the other 3 will consume at least 1 value), the maximum storage variation occurs, if the first 7 assignments generate one new value each, without consuming any, and all the consumptions occur later. Knowing from Step 2 that the memory size is 10 at the beginning of loop 2, it follows that the upper-bound of the memory size for this loop is 10+7=17 locations. With similar reasoning, one can conclude that during the execution of this loop, the memory trace could not go below 8 locations (see Figure 5c). Thus, the bounding rectangle for this loop has the upper edge 17, and lower edge 8.

4.4 Determining the memory size range

The memory requirement for a specification is the peak of the memory trace. Since the peak of the trace is contained within the bounding rectangles (along with the whole memory trace), the highest point among all the bounding rectangles represents an upper-bound of the memory requirement. For our illustrative example, the memory requirement will not exceed 17 - the highest upper-edge of the bounding rectangles (see Figure 5c).

Since the memory size at the boundaries between the loop nests is known, the memory requirement will be at least the maximum of these values. The maximum memory size at the boundary points thus represents the lower-bound of the memory requirement. For our illustrative example, the memory requirement is higher than 15 (the lower dotted line in Figure 5c), which is the maximum of the values at the boundaries of the three loops (Figure 5a). Therefore, the actual memory requirement will be in the range [15..17]. This memory requirement range represents the result of the first pass of the algorithm. The last step of our algorithm decides whether a more accurate approximation is necessary, in such case initiating an additional pass.

4.5 Improving the estimation accuracy

If the current estimation accuracy is not satisfactory, for each loop nest whose rectangle may contain the memory trace peak (the upper_edge higher than the previous memory lower-bound), a split of the iteration space is performed (by gradually splitting the range of the iterators, thus fissioning the loop nest). The *critical* rectangles corresponding to these loop nests will be replaced in a subsequent pass of the estimation algorithm by two new rectangles covering a smaller area (Figure 6a) and, therefore, following more accurately the actual memory trace, thus yielding a more accurate memory behavior analysis, and a more exact estimation. This process is iteratively repeated until a convenient estimation accuracy is reached. Figure 6a shows the refined bounding area, and 6b presents the actual memory trace for our example.

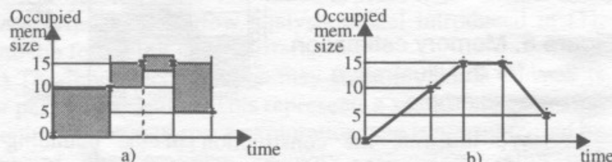

Figure 6. a) Accuracy refinement
b) complete memory trace

5. Discussion on parallelism vs. memory size

In the following we present a more complex example (Figure 7), which shows the utility of the estimation tool. This code contains a *forall* loop and multiple instructions executed in parallel (using *cobegin-coend*).

By analyzing the memory behavior of this code assuming the *forall* loop actually as a simple *for*, executed sequentially (we did not replicate the code due to lack of space), we obtain the behavior in Figure 8, and after the second pass we determine that the memory size is between [396..398].

By allowing the *forall* loop to be executed in parallel, the memory behavior becomes the one depicted in Figure 9, and the memory size is 300. Thus, the memory size for the parallel version of the code is 25% less than for the sequential case.

Having the parallel version require 25% less memory than the sequential one is a surprising result, since common sense would suggest that more parallelism in the code would need more memory. We have done some preliminary experiments in this direction, and they all seem to imply that more parallelism does not necessarily mean more memory. Moreover, we could not find (or produce) any example where the most parallel version of the code needs more memory than all the sequential versions. For most of the examples, the most parallel version had less memory requirement than most of the sequential ones. A possible explanation could be the fact that when instructions are executed in parallel, values are produced early, but also consumed early, and early consumption leads to low memory requirement. We intend to continue our work in this direction.

```
for I=0 to 97 do
  a[2*I, 0] = 0; a[2*I+1, 0] = 1;
  for J=0 to 99 do cobegin
    a[2*I,J+1]=a[2*I+1,J];
    a[2*I+1,J+1]=a[2*I,J];
  coend;
forall I=98 to 99 do
  a[2*I, 0] = 0; a[2*I+1, 0] = 1;
  for J=0 to 99 do cobegin
    a[2*I,J+1]=a[2*I+1,J];
    if(I!=99) a[2*I+1,J+1]=a[2*I,J]
    else a[2*I+1,J+1]=a[2*I,J]+a[2*I-2,J];
  coend;
for J=0 to 99 do
  for I=0 to 99 do cobegin
    a[2*I,J+101]=a[2*I+1,j+100];
    a[2*I+1,J+101]=a[2*I,J+100]+
                   a[2*I,J+99];
  coend;
```

Figure 7. Input specification with parallel instructions

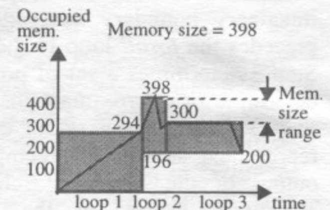

Figure 8. Memory behavior for example with simple for loop

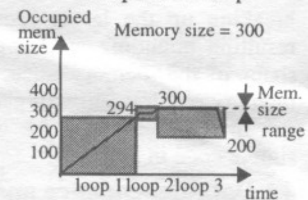

Figure 9. Memory behavior for example with *forall* loop

6 Experimental results

We compared our algorithm against a memory estimation tool based on symbolic execution, which assigns the signals to memory on a scalar basis to maximize sharing. We ran both algorithms on a SPARCstation 5, on 7 application kernels: image compression algorithm (Compress), linear recurrence

solver (Linear), image edge enhancement (Laplace), successive over-relaxation algorithm [8] (SOR), two filters (Wavelett, Low-pass), and red-black Gauss-Seidel method [8]. These examples are typical in image and video processing. Some of them (e.g., SOR, Wavelett, G-S) exhibit complex affine indices and conditionals.

The results of our tests are displayed in Table 1. Columns 2 and 3 show the memory requirement and the computation time obtained with the symbolic execution method. This memory size represents the optimal in terms of locations sharing. Column 4 represents the memory size estimate obtained with our algorithm (note that this is a lower-bound for the optimal value). For all the examples, the memory prediction is very close to the optimal value from column 2. The upper bound from Column 5 is used mainly as a measure of the confidence in the prediction. If this value is closer to the lower-bound estimate, the prediction is more accurate. Column 6 represents the percentile range:

100*(upper_bound - lower_bound)/upper_bound.

Table 1: Experimental results

Application	Symbolic exec.		MemoRex algorithm			
	Optimal mem. size	Time [s]	**Mem. estimate (l-b)**	Upper-bound	%	Time [s]
Compress	**10000**	133	**10000**	10000	0	0.01
Linear	**20002**	389	**20002**	20002	0	0.02
Laplace	**10404**	87	**10404**	10404	0	1
SOR	**38265**	205	**38265**	38265	0	0.01
Wavelett	**48003**	1265	**48002**	72002	33	0.02
			48002	54002	11	0.15
			48002	51002	5	0.42
Low-pass	**10100**	384	**10099**	10297	1	0.27
Gauss-Seidel	**13870**	197	**13870**	13917	~0	5

The lower this value, the more confidence we can have in the estimation. For most applications, high confidence is obtained within very reasonable time. A 0 percentile range means that the lower-bound is equal to the upper-bound, producing the exact value. When there is some slack between the two, the optimal value is usually very close to the lower-bound (the memory estimate), but in worst case, depending on the memory behavior complexity, it can be anywhere within that range.

For the Wavelett example, even though we obtained a very good estimate (48002 vs. 48003) from the first pass, we needed 3 passes to reduce the percentile range and increase the confidence in the estimation, due to the complexity of the memory behavior (the memory trace inside the loops is very irregular).

7 Conclusions

We presented a technique for estimating the memory size for multidimensional signal processing applications, as part of a design space exploration environment. Different from previous approaches, we have addressed this problem considering that the algorithmic specifications written in a procedural style may also contain explicit parallel constructs. Even if the initial input does not contain explicit parallelism, partitioning and design space exploration may introduce explicit parallelism in an attempt to achieve higher performance. Our method, based on an analysis of the memory size behavior, takes into account that signals with non-overlapping lifetimes can share the same storage locations (we assume that the assignment of arrays to the memory space is done at a later stage, and accounts for such sharing, e.g., [3]). Our experiments on typical video and image processing kernels show close to optimal results with very reasonable time. More details are available in [4].

We have also discussed some preliminary results on the influence of parallelism on the memory size, and shown that even though (parallelizing) transformations entail large variations, more parallelism does not necessarily mean more memory locations. A surprising result is that often the opposite is true. We plan to continue our research in this direction.

References

[1] F. Balasa, F. Cathoor, H. De Man, "Background memory area estimation for multidimensional signal processing systems," *IEEE Trans. on VLSI Systems*, Vol. 3, No. 2, pp. 157-172, June 1995.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic Program Parallelization," *Proc. of IEEE*, Vol. 81, No. 2, Feb. 1993.

[3] E. De Greef, F.Catthoor, H. De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems", *Proc. Intn'l. Conf. Application-specific Systems, Architectures and Processors*, pp. 66-75, 1997.

[4] P. Grun, F. Balasa, N. Dutt, "System-level memory size estimation," *University of California, Irvine,* Technical Report TR-97-37, Sept. 1997.

[5] F. Kurdahi, A. Parker, "REAL: A program for register allocation," *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 210-215, June 1987.

[6] P. Panda, N. Dutt, A. Nicolau "Architectural Exploration and Optimization of Local Memory in Embedded Systems," *Proc. International Symposium on System Synthesis (ISSS'97),* Antwerp, September 1997.

[7] P.E.R Lippens, J.L.Van Meerbergen, W.F.J. Verhaegh, A. van der Wef, "Allocation of multiport memories for hierarchical data streams," *Proc. IEEE Int. Conf. Comp.-Aided Design,* pp. 728-735, Santa Clara CA, Nov. 1993.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "*Numerical Recipes in C: The Art of Scientific Computing,*" Cambridge University Press, 1992.

[9] L. Ramachandran, D. Gajski, V. Chaiyakul, "An algorithm for array variable clustering," *Proc. European Design and Test Conf.*, pp. 262-266, Paris, France, March 1994.

[10] H. Schmit, D. E. Thomas, "Synthesis of application-specific memory designs," *IEEE Trans. on VLSI Systems,* Vol. 5, No. 1, pp. 101-111, March 1997.

[11] P. Slock, S. Wuytack, F. Catthoor, G. de Jong, "Fast and extensive system-level memory exploration for ATM applications," in *Proc. of 10th Intn'l Symp. on System-Level Synthesis,* Antwerp, Belgium, Sept. 1997.

[12] M. van Swaaij, F. Franssen, F. Catthoor, H. De Man, "High-level modeling of data and control flow for signal processing systems," in *Design Methodologies for VLSI DSP Architectures and Applications,* M. Bayoumi (ed.), Kluwer, 1993.

[13] I. Verbauwhede, C. Scheers, J. Rabaey, "Memory estimation for high-level synthesis," *Proc. 31st Design Automation Conf.*, pp. 143-148, San Diego CA, June 1994.

[14] M. Wolfe, *"High Performance Compilers for Parallel Computing,"* Addison-Wesley, Redwood City, CA, 1996.

[15] V. Zivojnovic, J. Martinez, C. Schlager, H. Meyr, "DSPStone: A DSP-oriented benchmarking methodology," *Proc. of ICSPAT'94*, Dallas TX, Oct. 1994.